Semantics-Aware Event Data Transformation for Process Mining

Inauguraldissertation
zur Erlangung des akademischen Grades
eines Doktors der Naturwissenschaften
der Universität Mannheim

vorgelegt von

Adrian Gianluca Rebmann
aus Zweibrücken

Mannheim, 2024

Dekan: Professor Dr. Claus Hertling, Universität Mannheim
Referent: Professor Dr. Han van der Aa, Universität Wien
Korreferent: Professor Dr. Stefanie Rinderle-Ma, Technische Universität München
Korreferent: Professor Dr. Heiner Stuckenschmidt, Universität Mannheim

Tag der mündlichen Prüfung: 11. Juni 2024.

Abstract

Process mining comprises methods to analyze organizational processes based on event data recorded by information systems during process execution. These methods generate actionable insights into how processes are truly executed and thereby support their improvement. However, the characteristics of event data that is available in organizations often differs from the data needs of process mining analyses. Specifically, unavailable data limits process analysis options, overly fine-granular data leads to uninformative process mining results, and inaccurate data even leads to incorrect results that do not mirror reality. These problems severely impact the opportunities and outcomes of process mining analyses. The goal of this doctoral thesis is to support organizations in overcoming these problems so that they can analyze their processes effectively using the event data available to them. Its main contributions are five approaches that automatically transform event data so that its characteristics satisfy the data needs of particular process analysis purposes. Specifically, we propose approaches to (1) annotate event data with semantic components to enable semantics-aware process analysis, (2) abstract fine-granular event data while adhering to user-defined requirements to enable purpose-driven process analysis, (3) transform user interaction data to task-level events to enable process analysis, (4) extract object-related information from event data to enable object-centric process analysis, and (5) detect best-practice violations in event data to provide insights into data-quality and conformance issues. The common driver of these approaches is the consideration of the semantics, i.e., the meaning, of events. We demonstrate the efficacy of the proposed approaches through quantitative evaluations using data obtained in real-world settings. Furthermore, we present application scenarios that underscore the usefulness of our approaches and highlight the analysis opportunities they enable for organizations.

Zusammenfassung

Process Mining umfasst Methoden zur Analyse von Unternehmensprozessen auf der Grundlage von Ereignisdaten, die von Informationssystemen während der Prozessausführung aufgezeichnet werden. Diese Methoden liefern Erkenntnisse darüber, wie die Prozesse tatsächlich ausgeführt werden und unterstützen damit die Prozessverbesserung. Die Eigenschaften verfügbarer Ereignisdaten unterscheiden sich jedoch häufig von den Anforderungen von Process-Mining-Methoden. Insbesondere schränken nicht verfügbare Daten die Möglichkeiten der Prozessanalyse ein, zu feingranulare Daten führen zu wenig aussagekräftigen Ergebnissen und ungenaue Daten führen sogar zu falschen Ergebnissen, die die Realität nicht widerspiegeln. Diese Probleme beeinträchtigen die Analysemöglichkeiten und -ergebnisse erheblich. Ziel dieser Dissertation ist es, Unternehmen bei der Überwindung dieser Probleme zu unterstützen, damit sie ihre Prozesse mit den Ereignisdaten, die ihnen zur Verfügung stehen, effektiv analysieren können. Der Forschungbeitrag besteht aus fünf Ansätzen, die Ereignisdaten automatisiert transformieren, um so die Lücke zwischen den Dateneigenschaften und den Datenanforderungen bestimmter Prozessanalysen zu schließen. Konkret schlagen wir Ansätze vor, um (1) Ereignisdaten mit semantischen Komponenten zu annotieren, die semantische Prozessanalysen ermöglichen, (2) feingranulare Ereignisdaten zu abstrahieren und dabei benutzerdefinierte Anforderungen einzuhalten, um zielgerichtete Prozessanalysen zu ermöglichen, (3) Benutzerinteraktionen zu Prozessereignissen zu abstrahieren, die Analysen auf Prozessebene ermöglichen, (4) Informationen über einzelne Objekte und deren Beziehungen aus Ereignisdaten zu extrahieren, um objektzentrierte Prozessanalysen zu ermöglichen und (5) Best-Practice-Verstöße in Ereignisdaten zu erkennen, die Einblicke in die Datenqualität und Konformitätsprobleme geben. Diese Ansätze haben gemein, dass sie die Semantik, d.h. die Bedeutung, von Ereignissen berücksichtigen. Wir zeigen die Wirksamkeit unserer Ansätze durch quantitative Evaluationen anhand von realen Ereignisdaten. Darüber hinaus stellen wir Anwendungsszenarien vor, die den Nutzen und die praktische Relevanz der Ansätze, sowie die damit verbundenen Analysemöglichkeiten für Unternehmen demonstrieren.

Acknowledgements

Writing this thesis would not have been possible without the encouragement and support of others. I would therefore like to express my sincere thanks to all those who were directly and indirectly involved in the creation of this thesis.

First and foremost, I would like to thank my supervisor, Han van der Aa, for his continuous support and feedback throughout my PhD time. I want to thank him for choosing me as his first PhD student, for helping me get started with a first project, for giving me the freedom to work on my own research topics and develop my ideas, for the close collaboration on our papers, and for the many hours he invested in me, especially during paper-writing phases. I learned a lot from him, for which I am truly grateful.

Secondly, I would like to thank my friend Jana Rehse. Without her encouragement I would not have made the move to Mannheim. I cannot thank her enough for all the advice—both professional and personal—that she has given me over the years. I owe a lot of this to her.

During my time as a PhD student, I had the privilege and pleasure to work with great people. I want to thank Matthias Weidlich, Timotheus Kampik, Carl Corea, and again Jana for being co-authors of some of the works included in this thesis. Thanks also to Timotheus for helping to make my research visit at SAP Signavio possible. I also want to thank Henrik Leopold, who, together with Han, organized the "Process Analytics Research Seminar". Taking part in this was a real highlight during my PhD time. Furthermore, I am grateful to my colleagues in the Process Analytics Group, Alexander Kraus and Robert Blümel, as well as to Diana Sola, Peter Pfeiffer, Luka Abb, and Michael Grohs for the great times we had at conferences, their support, and all the discussions on research-related and not-so-much-research-related matters.

I am immensely grateful to my friends and family, especially my mother, father, sisters, and grandparents, for supporting me throughout this time but also my entire life. Last, but most importantly, I want to thank my wife, Verena, for always believing in me, for her patience, and the sacrifices and compromises she made that allowed me to pursue this. None of it would have been possible without her.

Contents

Acronyms

BERT Bidirectional Encoder Representations from Transformers.

BPI Business Process Intelligence.

BPM Business Process Management.

BPMN Business Process Model and Notation.

CRF Conditional Random Field.

DFG Directly-Follows Graph.

ERP Enterprise Resource Planning.

HMM Hidden Markov Model.

LPM Local Process Model.

LTL Linear Temporal Logic.

MIP Mixed-Integer Programming.

NER Named-Entity Recognition.

NLP Natural Language Processing.

POS Part of Speech.

RPM Robotic Process Mining.

SRL Semantic Role Labeling.

UI User Interaction.

UML Unified Modeling Language.

URL Uniform Resource Locator.

XES eXtensible Event Stream.

XOC eXtensible Object-Centric Event Log.

List of Algorithms

List of Figures

List of Tables

Chapter 1

Introduction

This chapter provides an introduction to this doctoral thesis. Section 1.1 first outlines the context, before Section 1.2 motivates the need for transforming event data for process mining. Section 1.3 gives an overview of the contributions of this thesis. Section 1.4 describes the methodological background for the conducted research and Section 1.5 presents the publications that resulted from it. Finally, Section 1.6 outlines the remaining chapters of this thesis.

1.1 Context

Process mining enables the data-driven analysis of organizational processes based on event data that is recorded by information systems. In this context, an organizational process is "a collection of inter-related events, activities, and decision points that involve a number of actors and objects, which collectively lead to an outcome that is of value to at least one customer" [72]. Process mining offers actionable insights into how such a process is really executed. This includes the discovery of process models, the identification of deviations between actual and expected process behavior, and the diagnosis and forecasting of performance and compliance issues, all with the aim to support the improvement of a given process [8].

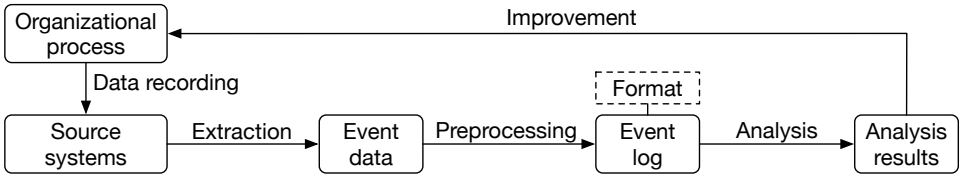


Figure 1.1: Overview of process mining.

Figure 1.1 shows an overview of how to get from the execution of an organizational process to analysis results that help improve this process. Data about the process execution is recorded and entered into source systems, such as *Enterprise Resource Planning* (ERP) systems, databases, transaction logs, message-oriented middleware, and document management systems [7]. Event data is extracted from one or more such source systems, preprocessed, e.g., filtered and cleaned, and stored in an event log that can be used for process mining [73]. The event log has a certain format that defines entity and relation types and their attributes, which are used to represent the process executions.

Table 1.1 shows an event log that captures two executions of a request-handling process. Each event corresponds to an activity that was performed in the process, the timestamp of its occurrence, a resource that performed the corresponding activity, and an identifier of the request it belongs to. As such, the data captures the order of steps that were performed to handle each request. Take, for instance, *request-1*: after receiving the request, it was first categorized. Subsequently, a casual examination took place and, upon reaching a decision, the request was accepted. The requester was then notified about this outcome and the request was archived.

Table 1.1: Event log capturing two executions of a request-handling process.

Request ID	Activity	Timestamp	Resource
request-1	Request received	05-20 09:07	System
request-1	Categorize request	05-20 09:11	User1
request-2	Request received	05-20 10:23	System
request-2	Categorize request	05-20 10:55	User1
request-1	Examine casually	05-20 11:24	User3
request-1	Accept request	05-20 11:41	User3
request-1	Inform requester	05-20 11:52	User1
request-1	Archive request	05-20 11:53	User1
request-2	Examine thoroughly	05-20 13:41	User2
request-2	Reject request	05-20 15:02	User4
request-2	Archive request	05-20 15:13	User1
request-2	Inform requester	05-20 15:15	User1

The specific characteristics of an event log depend on the different aspects involved in its creation. The granularity of the events depends on the data recording and the source systems that are in place. The behavior that an event log captures depends on the process itself but also on what (part of the) process behavior is recorded and extracted. The entities and relations of the process that an event log captures depend on the choice for a specific log format and, finally, available event attributes depend on the extraction procedure.

1.2 Motivation

An event log can serve as a basis for a broad range of process mining analyses. However, its characteristics may also differ from the data needs of a given analysis purpose. This misalignment can manifest itself in three sorts of problems.

Unavailable data limits analysis options. Data that is not readily available in an event log may prevent an organization from performing certain process analyses. In particular, certain techniques consider the meaning of events and their data attributes. This involves recognizing what underlying activities mean in relation to the process, or what event attributes say about the performed step. Examples for this include social network analysis [12], which considers the actors that perform events, object-centric process analysis [6], which considers the different objects involved in a process, and semantic anomaly detection [5], which detects anomalous behavior by considering the meaning of activities. However, the information required to conduct these analyses, such as the actions, objects, and actors associated with events, is not readily available in most event logs. This issue considerably limits the analysis options and thus the insights that an organization can obtain about their process based on a given event log.

Overly fine-granular data leads to uninformative results. Data that is too fine-granular can lead to analysis results that do not provide useful insights into the process. In particular, fine-granular events typically cause a considerable amount of different activities and a high degree of behavioral variability in the event data. This leads to process mining results that are complex and challenging to interpret [187]. Consequently, an organization cannot improve its understanding of their process, which is a primary goal of applying process mining in the first place [187]. This problem is amplified if fine-granular events lack activity-level information and an explicit relation to a process [60]. For instance, if events are recorded at the granularity of individual user interactions such as *click button* or *input text*, their relation to process-related activities is unclear. When applying process mining to such data, the results are not only complex, but also lack a direct connection to an organizational process, thereby failing to improve our understanding of it.

Inaccurate data leads to incorrect results. Data that misrepresents the true process can lead to analysis results that are incorrect. This is particularly problematic considering that process mining aims to generate actionable insights about a given process. Organizations may derive incorrect conclusions, redesign their process based on them, and decrease their process quality rather than increasing it [126]. Such incorrect analysis results can result from the choice of a log format that cannot capture complex interrelations among multiple objects involved in a process. In particular, this can lead to false statistics about the process and process behav-

ior that diverges from reality [6]. Incorrect analysis results may also arise from data corresponding to anomalous process behavior, such as cancelled orders that were never created. Whether due to erroneous data recording or process execution issues, unknowingly including such behavior in an analysis can distort its results.

Although these problems severely impact the opportunities and outcomes of process analysis, there is little support for organizations to address them effectively. We aim to overcome this lack of support by automatically transforming event data so that its characteristics satisfy the data needs of particular analysis purposes.

1.3 Contributions

This thesis addresses the need for automated event data transformation to enable organizations to effectively analyze their processes based on event data available to them. Its main contributions revolve around the definition of five approaches that transform event data so that the characteristics of the transformation result satisfies the data needs of particular analysis purposes.

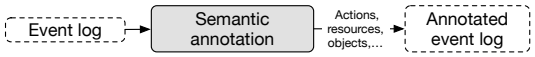
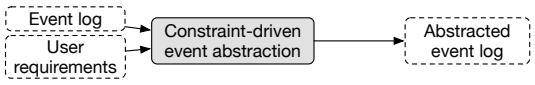
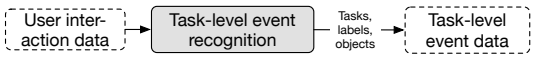
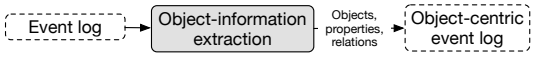
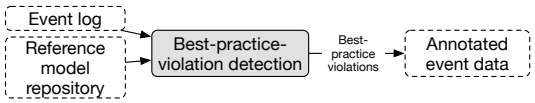
Goal	**Illustration**
1. Annotate an event log with semantic components to enable semantics-aware analysis.	
2. Abstract an event log while adhering to user-defined requirements to enable purpose-driven analysis.	
3. Transform user interaction data to task-level events to enable process analysis.	
4. Extract object-related information from an event log to enable object-centric analysis.	
5. Detect best-practice violations in an event log to provide insights into data-quality and conformance issues.	

Table 1.2: Overview of the approaches presented in this thesis.

As shown in Table 1.2 each approach takes event data as input and transforms it into a representation that enhances process analysis options. Our choice for the specific transformation use cases is motivated by the practical relevance of the anal-

ysis purposes they enable and because addressing them requires novel conceptual developments that benefit from the consideration of the semantics of event data. In this thesis, we consider the term *semantics* in a linguistic sense, i.e., it refers to the meaning of words [101]. As such, our approaches take the *meaning* of event data into account. The five approaches correspond to the following five contributions.

1. *Semantic annotation of event logs.* Process mining approaches often analyze event data in an abstract manner, without considering the meaning of the underlying activities. Some analysis techniques do require specific information, though. For instance, resource information is needed for handover-of-work analysis and information about the type of action executed in an event is needed to assess the importance of the underlying activity in the context of the process. However, such information is often not readily available in event logs. It is rather recorded implicitly as part of unstructured attributes such as the event's activity label or in unclearly named attributes or not captured by the data at all. In Chapter 3, we address this problem by proposing an approach that identifies and categorizes semantic components in events and creates an annotated event log that makes them explicit. In this manner, the approach enables a broad range of semantics-aware analysis scenarios.
2. *Constraint-driven abstraction of event logs.* Events that are recorded by information systems are often too fine-granular to be used for process analysis directly. When such data is used for process mining, e.g., discovery, this can lead to results that are hard to interpret due to a high complexity of the output. A range of event-abstraction approaches have been proposed [187] that aim to tackle this problem. However, none of these allow a user to specify what the resulting (abstracted) event log should look like. This is crucial if they want to perform process analysis that has specific data needs, though. For instance, a user who wants to analyze an event log with respect to the resources that perform activities needs to be sure that each activity in the abstracted log was performed by a single resource. In Chapter 4, we propose an event-abstraction approach that allows for this, by letting a user impose requirements on the resulting log in terms of constraints. As such, the approach supports the specification of properties that the abstracted log should adhere to, so that it can be used for a specific analysis purpose.
3. *Task-level-event recognition from user interaction data.* User interaction data provides detailed records about how users perform their tasks in a process, even when performing them across different applications. Although their comprehensiveness provides a promising basis for process mining, user interaction events cannot be used directly for this purpose, because they do not meet two essential requirements. In particular, they neither indicate their

relation to a process-level activity nor their relation to a specific process execution. Therefore, user interaction data needs to be transformed so that it meets these requirements before process mining techniques can be applied. In Chapter 5, we propose an unsupervised approach for recognizing task-level events from user interaction data that addresses this transformation problem. In this manner, our approach creates events from low-level data that can be used in process mining settings.

4. *Object-information extraction from event logs.* Process mining techniques have long been developed with the assumption that each process step and, thus, event, can be associated with a single process execution, a so-called *case* [8]. However, in real-world processes, individual steps often relate to multiple objects that have interrelations across process executions, such as multiple customer orders that are shipped in a single package [6]. Therefore, object-centric event logs have recently been introduced, which allow events to relate to any number of objects and enable object-centric analyses. Event data of multi-object processes is often only available in a case-centric format, though, without access to the systems from which it was extracted. This format obscures the true relations between objects and associated events, causing data quality issues that lead to incorrect analysis results. In Chapter 6, we propose an approach that addresses this problem by transforming case-centric event logs into object-centric ones, based on object information it extracts from its input. As such, the approach enables organizations to use their existing case-centric event logs for object-centric process mining.
5. *Best-practice-violation detection in event logs.* Detecting undesired behavior in event logs can reveal data-quality and conformance issues. It typically requires dedicated process models that specify what desired behavior entails [41]. Such models are rarely available, though, and their creation involves substantial effort [72]. *Reference process models* [80] serve as best-practice templates for organizational processes across domains and can, thus, mitigate the need for dedicated models by providing a basis to check for undesired behavior. However, matching a single reference model to a real-world event log is impractical because organizational needs can vary (despite similarities in process execution) and because event logs may cover the behavior of multiple reference models. In Chapter 7, we propose an approach that still allows for detecting undesired process behavior based on such models. It extracts best-practice behavior from a reference model repository, selects relevant behavior for a given event log, and identifies violations from it. The resulting best-practice violations provide insights into potential data quality issues and conformance problems.

1.4 Methodological Background

The research presented in this thesis is conducted in the field of process mining. Because we develop technical approaches that tackle real-world problems, our research falls into the realm of *algorithm engineering*. Mendling et al. [128] propose a framework for algorithm engineering research that applies to process mining research on the development of technical approaches such as those proposed in this thesis [37]. This framework provides ontological, epistemological, and methodological perspectives.

Ontological perspective of algorithm engineering. As shown in Figure 1.2, the framework identifies four ontological entities in algorithm engineering: the *real-world problem*, *algorithmic task*, *algorithm design*, and *algorithm implementation*.

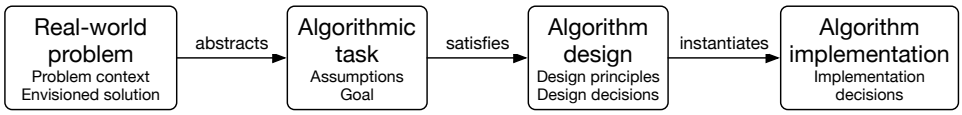


Figure 1.2: Ontological perspective of algorithm engineering. Adapted from [128].

Real-world problem. Algorithm engineering is motivated by real-world problems, which exist in a specific context and hint at an algorithm as part of an envisioned solution [128]. When developing our approaches, we always start from a real-world problem. For instance, for our constraint-driven abstraction approach, we started from the problem that users cannot express their requirements for the characteristics of abstracted event logs.

Algorithmic task. An algorithmic task abstracts from a real-world problem and conceptualizes it, capturing essential assumptions and requirements. It has a goal that specifies what the algorithm should achieve regarding output and performance [128]. For each real-world problem we address in this thesis, we derive one or more algorithmic tasks that jointly address it. For instance, for constraint-driven abstraction, we derived three algorithmic tasks to address the corresponding problem: (1) grouping events while adhering to constraints, (2) finding an optimal solution from a set of possible groupings, and (3) abstracting an input log based on a grouping.

Algorithm design. An algorithm design incorporates design principles and decisions to address (satisfy) an algorithmic task [128]. In this thesis, we propose approaches comprising algorithm designs that satisfy the tasks we derived from a real-world problem. For instance, to satisfy the tasks of constraint-driven abstraction, we designed algorithms for exhaustive and heuristic event grouping, adopt an algorithm for solving optimal grouping tasks, and design an abstraction algorithm.

Algorithm implementations. An algorithm implementation instantiates an algorithm design, which requires implementation decisions. Executing an implementation on data from the real-world problem generates specific output and gives an indication of the algorithm's empirical performance [128]. We provide implementations of all our algorithm designs in Python, which we use to gain knowledge about them through evaluation experiments.

Mendling et al.'s framework describes different knowledge types that can be gained in algorithm engineering in its epistemological perspective, which we outline next.

Epistemological perspective of algorithm engineering. In its epistemological perspective, the framework [128] distinguishes between knowledge *of* and *about tasks* as well as knowledge *of* and *about designs*. The approaches proposed in this thesis contribute to the *knowledge of design* through novel designs that satisfy algorithmic tasks that we derived from real-world problems. By assessing our approaches in evaluation experiments, we also contribute to *knowledge about design*.

Knowledge of design. Knowledge of design refers to knowledge of algorithms that exhibit enhanced performance in crucial dimensions like execution time or output accuracy compared to other algorithms.

Our approaches improve over existing designs in various manners. Using state-of-the-art techniques from Natural Language Processing [100], our semantic-annotation approach improves over existing work for semantic event parsing in both scope and accuracy. No previous event-abstraction approach allows a user to define requirements on the output, which our approach improved. Existing approaches do not provide an end-to-end solution to transform user interactions into events that can be used for process mining, which our approach for recognizing task-level events improves. Similarly, existing techniques for detecting undesired process behavior typically require a dedicated normative model as input, which our best-practice-violation-detection approach alleviates through the use of reference models. Lastly, our object-information-extraction represent the first approach to solve its algorithmic tasks. It does so, by combining and enhancing existing algorithm designs from Natural Language Processing and data profiling [14].

Knowledge about design. Knowledge about design encompasses insights about properties and characteristics of algorithm designs. Among others, such knowledge relates to a design's *performance* and its *sensitivity*. *Performance knowledge* focuses on if a designs meets the requirements of its task. It can inquire about satisfaction, i.e., if the design satisfies the task requirements, or about the degree of satisfaction, i.e., to which extent the design is better in satisfying the task requirements than others. *Sensitivity knowledge* focuses on the design's robustness in performance when facing changes, e.g., across parameter settings [128].

Through experimental evaluations using implementations of our designs, we contribute to *performance knowledge* and *sensitivity knowledge*. We assess the

performance of our designs using collections of real-world event logs and established performance metrics that are specific to the given task and compare our work against baselines and state-of-the-art approaches. For instance, we assess how accurately our semantic-annotation approach identifies semantic components from textual labels and compare it against a state-of-the-art label parser for this task. We provide *sensitivity knowledge* by systematically varying parameter settings of our approaches. For instance, our approach for detecting best-practice-violations uses a similarity threshold that determines when two words are considered similar. When assessing our approach, we compare various configurations of this setting.

Methodological perspective of algorithm engineering. From a methodological point-of-view, Mendling et al. [128] emphasize that various validity concerns must be considered when evaluating process mining algorithm designs. We account for these validity concerns in the following manner.

Logical validity. Logical validity refers to why an evaluation goal is valid [128]. We make sure to base the formulation of the evaluation goal on formal or empirical knowledge of prior research. For instance, when evaluating our constraint-driven-abstraction approach, we build on the empirical knowledge that the complexity of a process model correlates with its comprehensibility [159]. We thus set the goal to show that the abstracted logs lead to the discovery of less complex process models than the logs that result from applying baseline techniques for the problem at hand.

Internal validity. The design of an evaluation is internally valid if its manipulation causes an observed effect [128]. We account for *internal validity* by limiting the confounding variables in our experiments. In particular, we keep the execution environment unchanged across different runs of our approaches and any baselines. Furthermore, we separate training from test data in a randomized manner and conduct multiple evaluation runs (in a cross-validation setting), where applicable.

Implementation validity. Implementation validity refers to, among others, threats regarding alternative implementation options for a design. In particular, implementation decisions may turn into confounding variables [128]. To account for implementation validity, we implement all of our approaches in Python and make any code, evaluation data, evaluation scripts, gold standards, and raw results publicly available, ensuring reproducibility.

External validity. External validity measures how well findings generated from data used in evaluation experiments can be extended to other data [128]. To account for *external validity*, all of our proposed approaches are evaluated using real-world data sets that span a broad range of process domains and that are established in the process mining research community. Using data across process domains ensures the presence of diverse natural language attribute values including activities that range from the finance domain, e.g., *validate loan application* to healthcare

domains, e.g., *ultrasound configuration*, covering different labeling styles. Furthermore, the different event logs are recorded at various levels of granularity and the traces differ considerably in terms of variability.

Construct validity. Construct validity essentially considers if an evaluation measure appropriately measures the intended property [128]. To account for construct validity, we employ established evaluation metrics that measure the performance of our approaches with respect to the task they solve. For instance, to assess our object-information-extraction approach, we use the well-known *precision*, *recall*, and F_1*-score* metrics. In this context, precision is the fraction of objects extracted by our approach that correspond to actual objects contained in the gold standard and recall is the fraction of actual objects that are correctly extracted by our approach. Any gold standards that we use to assess our approaches either existed beforehand or are established by at least two researchers independently before discrepancies are discussed and consolidated.

Conclusion validity. Conclusion validity is concerned with the question to which extend conclusions that are drawn from evaluation results are reasonable for the evaluation goal [128]. To strengthen conclusion validity, we conduct quantitative experiments that use established evaluation measures and conduct additional qualitative evaluations. In particular, we apply our approaches to various application cases and report on an in-depth look into the results. Furthermore, we report on threats to validity where applicable.

This discussion of algorithm engineering research methodology in relation to the work presented in this thesis highlights that our contributions align with recognized research standards. Moreover, it underscores that we have made substantial contributions to the body of knowledge within the field of process mining.

1.5 Publications

The research conducted as part of this doctoral thesis resulted in six published papers and one paper that is currently under submission, which correspond to the contributions introduced in Section 1.3.

Semantic annotation of event logs.

- [149] **A. Rebmann**, H. van der Aa: Extracting semantic process information from the natural language in event logs. In: International Conference on Advanced Information Systems Engineering, 57–74 (2021)
- [148] **A. Rebmann**, H. van der Aa: Enabling semantics-aware process mining through the automatic annotation of event logs. Information Systems 102111 (2022)

Constraint-driven abstraction of event logs.

- [157] **A. Rebmann**, M. Weidlich, H. van der Aa: GECCO: constraint-driven abstraction of low-level event logs. In: IEEE International Conference on Data Engineering, 150–163 (2022)

Task-level-event recognition from user interaction data.

- [151] **A. Rebmann**, H. van der Aa: Unsupervised Task Recognition from User Interaction Streams. In: International Conference on Advanced Information Systems Engineering, 141–157 (2023)
- [150] **A. Rebmann**, H. van der Aa: Recognizing Task-level Events from User Interaction Data. Information Systems 102404 (2024).

Object-information extraction from event logs.

- [156] **A. Rebmann**, J. Rehse, H. van der Aa: Uncovering Object-centric Data in Classical Event Logs for the Automated Transformation from XES to OCEL. In: International Conference on Business Process Management (2022)

Best-practice-violation detection in event logs.

- [153] **A. Rebmann**, T. Kampik, C. Corea, H. van der Aa: Mining Constraints from Reference Process Models for Detecting Best-Practice Violations in Event Logs. Information Systems, under submission.

During his doctoral project, the author also contributed to research projects beyond the scope of this thesis, which has led to the publication of one journal article, one conference paper, two workshop papers, and two tool demonstration papers.

- [5] H. van der Aa, **A. Rebmann**, H. Leopold: Natural language-based detection of semantic execution anomalies in event logs. Information Systems 102, 101824 (2021)
- [155] **A. Rebmann**, P. Pfeiffer, P. Fettke, H. van der Aa: Multi-perspective identification of event groups for event abstraction. In: International Conference on Process Mining. Workshops (2022)
- [43] J. Caspary, **A. Rebmann**, H. van der Aa: Does this make sense? Machine learning-based detection of semantic anomalies in business processes. In: International Conference on Business Process Management (2023)

- A. Bergman, **A. Rebmann**, T. Kampik: BPMN2constraints: breaking down BPMN diagrams into declarative process query constraints. In: International Conference on Business Process Management. Demonstration Track (2023)
- A. Goossens, **A. Rebmann**, J. De Smedt, J. Vanthienen, H. van der Aa: From OCEL to DOCEL—Datasets and automated transformation. In: International Conference on Process Mining. Workshops (2023)
- F. Lang, D. Hida, Y. Bian, **A. Rebmann**, H. van der Aa: IM-Viz: A tool for the step-by-step visualization of the inductive miner. In: International Conference on Process Mining. Demonstration Track (2023)

1.6 Outline

The remainder of this thesis is structured into the following seven chapters, with the core chapters each introducing an approach.

- *Chapter 2: Background.* This chapter provides background information that is essential for the remainder of the thesis. It focuses on preliminaries of process mining and on Natural Language Processing.
- *Chapter 3: Semantic Annotation of Event Logs.* This chapter presents our approach for annotating event logs with components that are relevant for semantics-aware process analysis. In particular, our approach first identifies up to eight semantic components per event, revealing information about actions, object types, and resources, before it further categorizes identified actions and resources, which allows for an in-depth analysis of key process perspectives. An evaluation using a broad range of event logs shows the approach's efficacy, while application scenarios enabled by our approach highlight its usefulness. This chapter is based on concepts and results previously published in Information Systems [148] and the International Conference on Advanced Information Systems Engineering [149].
- *Chapter 4: Constraint-Driven Abstraction of Event Logs.* In this chapter, we present our constraint-driven event-log-abstraction approach that enables users to impose requirements on the output log. It groups events so that the requirements are satisfied and the behavioral distance to the input log is minimized. Since exhaustive event-log abstraction has exponential run time complexity, we also offer a heuristic approach guided by behavioral relations found in the log. We show that the abstraction quality of our approach is superior to baseline approaches and demonstrate the relevance of considering user requirements during event-log abstraction in real-world settings. This

chapter is based on concepts and results previously published in the IEEE International Conference on Data Engineering [157].

- *Chapter 5: Task-Level-Event Recognition from User Interaction Data.* In this chapter, we present an unsupervised approach for recognizing task-level events from user interaction data. It segments user interaction data to identify tasks, categorizes these according to their type, and relates tasks to each other via object instances it extracts from the user interaction events. In this manner, our approach creates events that meet the requirements of process mining settings. Our evaluation demonstrates the approach's efficacy and shows that its combined consideration of control-flow, data, and semantic information allows it to consistently outperform baseline approaches. This chapter is based on concepts and results previously published in Information Systems [150] and the International Conference on Advanced Information Systems Engineering [151].
- *Chapter 6: Object-Information Extraction from Event Logs.* This chapter presents our approach to extract object-related information from event logs and automatically transform the input event log into an object-centric format. It achieves this by combining the semantic analysis of textual attributes with data profiling and control-flow-based relation extraction techniques. We demonstrate our approach's efficacy through evaluation experiments and highlight its usefulness by applying it to real-world event logs in order to mitigate quality issues caused by their case-centric representation. This chapter is based on concepts and results previously published in the International Conference on Business Process Management [156].
- *Chapter 7: Best-Practice-Violation Detection in Event Logs.* In this chapter, we present an approach for detecting best-practice violations in event logs. The approach mines declarative best-practice constraints from a reference model collection, automatically selects constraints that are relevant for a given event log, and checks for constraint violations. We demonstrate the efficacy of our approach through an evaluation based on real-world process model collections and event logs. This chapter is based on concepts and results previously described in a manuscript that is under submission for Information Systems [153].
- *Chapter 8: Conclusion.* This chapter concludes this doctoral thesis. We summarize the main results and reflect on their implications for research and practice. Furthermore, we provide directions for future research.

Chapter 2

Background

This chapter provides the necessary background for the remainder of the thesis. Section 2.1 introduces the preliminaries of process mining and gives formal definitions of relevant process mining concepts that are used throughout this thesis. Section 2.2 introduces methods from Natural Language Processing, which our approaches use to incorporate considerations related to the semantics of event data.

2.1 Process Mining

Every organization, no matter whether it is a commercial enterprise, a government agency, a non-profit or a healthcare organization, is driven by processes. These processes organize how work is performed and comprise interconnected events, activities, decisions, actors, and objects that jointly produce an output of value [72]. The way in which processes are performed impacts the quality of their output, e.g., the quality of a service that should be delivered through a process, and the efficiency in which this output is generated. Therefore, managing and continuously improving these processes is vital for organizations to ensure high process efficiency and quality [72]. The discipline that studies how to support organizations in this regard is called *Business Process Management* (BPM). BPM combines methods to design, enact, control, and analyze organizational processes [96].

Initially, the main focus of BPM methods was on designing and enacting processes in a model-driven manner, without taking data about their actual execution into account [96]. Nowadays, the execution of organizational processes is supported by various types of information systems, such as enterprise systems and web applications. Using these systems during process execution leaves digital traces of the performed activities [7]. These traces can be extracted from the information systems in the form of event data. Based on this event data, *process mining* provides means to analyze organizational processes in a data-driven manner. Process

mining techniques generate insights into how processes are really executed, identify bottlenecks and deviations from expected process behavior, and predict and diagnose performance and compliance issues. These insights eventually support the improvement of organizational processes [8].

The remainder of this section provides the basic concepts related to process mining that are relevant in the context of this thesis. First, Section 2.1.1 introduces and defines event data, given that it is the main input to any process mining analysis. Then, Section 2.1.2 introduces and defines process models that are commonly used to represent process mining results and to specify normative process behavior that serves as input to process mining techniques alongside event data.

2.1.1 Event Data

Event data captures the individual steps that are performed during the execution of an organizational process. It is *the* main input to any process mining analysis [7]. Typically it is extracted from information systems, preprocessed, and then stored in an *event log* for post-hoc process analysis. Event logs can either be represented in a case-centric or an object-centric format. Certain situations require one to move from post-hoc to streaming process analysis instead. For instance, when an analysis goal requires immediate insights into the current state of a process based on event data. In such cases, events arrive in the form of an *event stream*, which has to be processed in an online manner.

In the following, we first define *events* as the basic entity of any form of event data representation, before defining *case-centric event logs*, *object-centric event logs*, and *event streams*.

Events. Events represent information about how an organizational process is executed. Each event refers to one step (activity) that is performed in the context of a process at a specific point in time. Events can have any number of attributes that capture their context, such as the resource that performed the step. In the context of this thesis, we formally define events and their attributes as follows.

Definition 1 (Events, Attributes) *Let $\mathcal{E}$ be the universe of all events. An event $e \in \mathcal{E}$ is a tuple $e = (a, ts, D)$; $a \in \mathcal{A}$ is the* activity *that e relates to, $\mathcal{A}$ being the universe of all activities, and $ts \in \mathcal{TS}$ is e's timestamp, $\mathcal{TS}$ being the universe of all timestamps. D is a set of data attributes that capture additional context for event e. We use $\mathcal{D} = D \cup \{a, ts\}$ to refer to all attributes of an event. We write $\mathsf{dom}(d)$ for the domain of the attribute $d \in \mathcal{D}$ and $\mathsf{name}(d)$ for its name.*

Note that, in the remainder, we use dot notation as a shorthand to refer to attributes of events, e.g., using $e.a$ as a shorthand to refer to the activity of an event $e \in \mathcal{E}$ and $e.d$ to refer to its value for attribute d.

Case-centric event logs. Case-centric event logs assume that each process execution is clearly distinguishable from others and can therefore be considered individually. For example, consider a request-handling process: for each request that is submitted, a number of activities, such as *categorize request* and *examine request*, are performed in coordination to eventually decide if a request is accepted or not. An exemplary case-centric event log of this process is depicted in Table 2.1.

Table 2.1: Case-centric event log with two cases of a request-handling process.

Trace	CaseID	Activity	Timestamp	Resource
σ_1	request-1	Request received	05-20 09:07	System
	request-1	Categorize request	05-20 09:11	User1
	request-1	Examine casually	05-20 11:24	User3
	request-1	Accept request	05-20 11:41	User3
	request-1	Inform requester	05-20 11:52	User1
	request-1	Archive request	05-20 11:53	User1
σ_2	request-2	Request received	05-20 11:23	System
	request-2	Categorize request	05-20 11:55	User1
	request-2	Examine thoroughly	05-20 13:41	User2
	request-2	Reject request	05-20 15:02	User4
	request-2	Archive request	05-20 15:13	User1
	request-2	Inform requester	05-20 15:15	User1

As shown, in a case-centric event log, each event belongs to precisely one *case* that corresponds to a single execution of a process, which is typically indicated by a case identifier (`CaseID`).

All events related to the same case are grouped into a *trace*, a sequence of events that are ordered by the time of their occurrence. As an example, consider the trace σ_1 in Table 2.1. It shows that the request was first received and categorized, before it was examined. Based on the result of the examination, the request was accepted and the requester was notified about this outcome. Finally, the request was archived. We formally define such traces as follows.

Definition 2 (Traces) *A trace $\sigma \in \mathcal{E}^*$ represents a single execution of an organizational process (a* case*) and consists of a finite sequence of events $\sigma = \langle e_1, ..., e_n \rangle$, with their order following from their timestamps. Given a trace σ, we use σ^a to refer to the sequence of its activities, i.e., $\sigma^a = \langle e_1.a, \ldots, e_n.a \rangle$; traces with equal activity sequences belong to the same trace variant.*

Note that, for illustration purposes, we commonly represent a trace's events by their activities in the coming chapters. For instance, we represent σ_1 as $\langle$*Request received*, *Categorize request*, *Examine casually*, *Accept request*, *Inform requester*, *Archive request*$\rangle$.

Having defined events and traces, we define *case-centric event logs* as follows.

Definition 3 (Case-centric event logs) *A (case-centric) event log $L \subseteq 2^{\mathcal{E}^*}$ is a finite multi-set of traces, where each trace corresponds to a case. L^a is the multi-set of activity sequences obtained from all traces in L, i.e., $L^a = \{\sigma^a \mid \sigma \in L\}$. We use $E_L \subseteq \mathcal{E}$ to denote the set of all events contained in L's traces, $A_L \subseteq \mathcal{A}$ to denote the set of activities that the events in E_L relate to, and $\mathcal{D}_L$ to denote the set of attributes that events in E_L have.*

Most of the approaches we propose in this thesis take an event log according to this definition as input and also produce such a log as their output.

Object-centric event logs. The assumption that process executions can be considered individually (i.e., as cases) does not hold for all types of organizational processes. In particular, it does not hold for processes that involve multiple objects of different types with complex interrelations. For example, in an order-handling process, orders can consist of multiple items and multiple orders can be shipped in one package. For such a process, there is no main object type, e.g., *order*, to serve as an unambiguous notion of a case. Therefore, event data of such processes are better represented as an object-centric event log that make objects, e.g., orders, items, and packages, and their interrelations explicit. In such a log, each event has a special attribute *objects* that contains references to any objects associated with the event. The objects themselves are stored separately along with their own properties. We formally define *objects* and *object-centric event logs* as follows.

Definition 4 (Objects) *Let $\mathcal{O}$ denote the universe of all objects. An object $o \in \mathcal{O}$ is a tuple $o = (oi, ot, vmap)$, with oi as its identifier, ot its type, and $vmap$ a value map, which captures the assignment of values to o's properties.*

Definition 5 (Object-centric event logs) *An object-centric event log is a tuple $OL = (O, E)$ that comprises a set of objects $O \subseteq \mathcal{O}$ and a set of events $E \subseteq \mathcal{E}$. Each event $e \in E$ has an attribute $d \in e.D$ with $\mathsf{name}(d) =$ objects, a map, which associates object types with identifiers of object instances, to which e relates. The events in E have a known partial order, following from their timestamps.*

An exemplary object-centric event log of an order-handling process is shown below consisting of a set of events (Table 2.2) and a set of objects (Table 2.3). The events correspond to the handling of two orders, involving several items that are shipped in multiple packages. As shown, a single event can relate to any number of objects of different types. For instance, event *e9* in Table 2.2 has relations to two orders, one package, two items, and one customer, i.e., $e9.objects = \{Order : \{o1, o2\}, Package : \{p2\}, Item : \{i1\_1, i2\_1\}, Customer : \{Pete\}\}$.

Table 2.2: Exemplary object-centric event log: events.

Event	Activity	Timestamp	Objects			
			Orders	Packages	Items	Customer
e1	Create order	05-20 09:07	{*o1*}	∅	{*i1_1*,*i1_2*}	{*Pete*}
e2	Reorder item	05-23 10:40	{*o1*}	∅	{*i1_1*}	{*Pete*}
e3	Pick item	05-23 14:20	{*o1*}	∅	{*i1_2*}	{*Pete*}
e4	Send package	05-23 17:26	{*o1*}	{*p1*}	{*i1_2*}	{*Pete*}
e5	Create order	06-03 19:17	{*o2*}	∅	{*i2_1*}	{*Pete*}
e6	Pick item	06-04 15:20	{*o1*}	∅	{*i1_1*}	{*Pete*}
e7	Update order	06-04 18:11	{*o2*}	∅	{*i2_1*}	{*Pete*}
e8	Pick item	06-05 11:48	{*o2*}	∅	{*i2_1*}	{*Pete*}
e9	Send package	06-06 16:20	{*o1*,*o2*}	{*p2*}	{*i1_1*,*i2_1*}	{*Pete*}

Table 2.3: Exemplary object-centric event log: objects.

Type	Object Instances
Customer	{(*Pete*,*Customer*, {})}
Order	{(*o1*,*Order*, {}), (*o2*,*Order*, {})}
Package	{(*p1*, *Package*, {Weight : 70.8}), (*p2*, *Package*, {Weight : 20.4})}
Item	{(*i1_1*, *Item*, {Weight:12.5}), (*i1_2*, *Item*, {Weight:70.8}), (*i2_1*, *Item*, {Weight:7.9})}

The objects themselves, including any object-specific properties, are stored separately. For instance, consider the item $i1\_1$ in Table 2.3 that event *e9* relates to, consisting of an identifier $o.oid = i1\_1$, a type $o.ot = Item$, and a value map $o.vmap = \{\texttt{Weight} : 12.5\}$ that captures its weight.

Event streams. In certain situations, post-hoc process analysis, for which previously recorded data is first extracted from systems and then stored in event logs, is not possible. On the one hand, certain analysis goals, such as process monitoring, require (near) real-time insights based on event data. On the other hand, when facing large data volumes, events simply cannot be processed all at once. Therefore, such situations demand a single-pass and online processing of event data [39]. In such cases, events are assumed to arrive in the form of an event stream that serves as input to so-called streaming process mining techniques. These handle events as they become available [39] and provide (near) real-time insights into process executions. We define event streams as follows.

Definition 6 (Event streams) *An event stream S_E is a potentially infinite sequence of events, i.e., $S_E \in \mathcal{E}^* \forall_{1 \leq i < j \leq |S_E|} S_E(i) \neq S_E(j)$.*

2.1.2 Process Models

While event data captures the real-world execution of organizational processes, process models serve as representations of these processes. There are different reasons for establishing models when managing and improving processes and these can take various forms depending on their purpose. An important purpose is the provision of a common process understanding among the people who are involved in its execution [72]. In process mining, process models are used to present the results of process discovery, where the aim is to identify a process model based on an event log [7]. Thus, they play a key role in analyzing the actual (as-is) behavior of a process, providing a basis for a comprehensive process understanding. Furthermore, process models are essential for specifying normative (to-be) process behavior in conformance checking, where the aim is to identify deviations from such behavior in an event log [41]. In our approaches and their evaluation, we also rely on process models for these purposes.

On a high level, there are two established paradigms of how to model processes: through imperative and declarative process models [160]. Both of these paradigms have their own benefits. For instance, imperative models are generally easier to understand for humans than declarative ones [92, 190], whereas declarative models are better suited to specify normative behavior of flexible processes and subsequently check for violations [58, 84]. We use both types of models in our work depending on their suitability for the respective transformation use case. Therefore, we introduce both of them in the remainder of this section.

Imperative process models

In imperative process models, every possible execution sequence of a process is explicitly modelled. As a consequence, anything that is not modeled is not allowed to happen in the process. Note that, for brevity, we refer to imperative process models as *process models* in the remainder, whereas for declarative ones, we explicitly write *declarative process models*.

In this thesis, we use a generic definition of *process models*, which is independent of a specific notation. It defines a process model as a set of allowed execution sequences and an assignment of roles to activities for which they are responsible.

Definition 7 (Process models, Process model collections) *Using $\mathcal{M}$ to denote a process model collection, each process model $M \in \mathcal{M}$ is a tuple $M = (A, F, R,$* `performedBy`*). A is the set of steps in M and, just like for events, we call $a \in A \subseteq \mathcal{A}$ an* activity. *F is the set of unique finite execution sequences allowed by M, with each $\pi \in F = \langle a_1, ..., a_n \rangle$ as a sequence of activities in A. Focusing on finite execution sequences captures the intuition that each process execution is*

expected to complete in a finite number of steps. Using $\mathcal{R}$ to denote the universe of all roles, $R \subseteq \mathcal{R}$ is a set of roles involved in the process represented by M. Finally, `performedBy` $: \mathcal{A} \to \mathcal{R} \cup \{\perp\}$ *is a mapping from activities to roles, e.g., a* create invoice *activity may be mapped to a* vendor *role in a procurement process model.*

There is a variety of (graphical) notations to model processes in an imperative manner, including Business Process Model and Notation (BPMN) [21] diagrams and directly-follows graphs (DFGs).

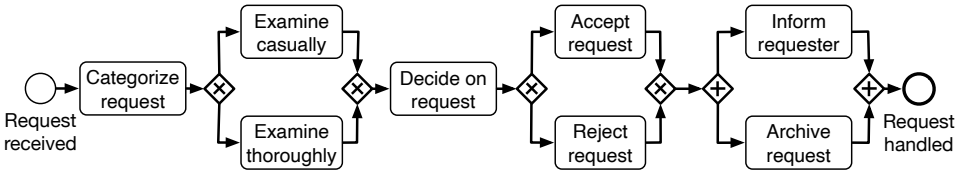


Figure 2.1: An exemplary BPMN diagram.

Figure 2.1 shows an exemplary BPMN diagram of a request-handling process. The model starts and ends with events, which are represented as circles, while activities, called *tasks* in BPMN, are represented as rectangles with rounded edges. The diamond-shaped symbols with a $+$ denote *parallel gateways*, whereas the ones with an $\times$ denote *exclusive gateways*. These gateways indicate concurrency and exclusive choices respectively. As such, the model shows that the process starts with the receipt of a request that is first categorized and then either checked casually or thoroughly. Afterwards, a decision about the request is made, after which it is either accepted or rejected. Finally, the requester is informed about the decision and the request is archived, which can be done in either order.

BPMN is a comprehensive notation with more than 100 different symbols that allow for modeling a broad range of processes in a detailed manner [72]. This also includes the representation of process perspectives beyond the control-flow, i.e., beyond the representation of ordering relations between activities. For instance, the resource perspective can be modeled using so-called *pools* and *lanes* to respectively represent organizational entities and roles [21].

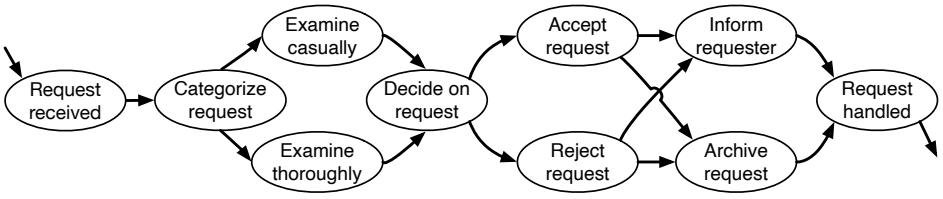


Figure 2.2: An exemplary DFG.

Figure 2.2 shows the same request-handling process modeled as a DFG. Such DFGs provide a first impression of a process and are straightforward to compute from an event log. In essence, a DFG indicates if two activities ever immediately succeed each other in the event log. We formally define them as follows.

Definition 8 (Directly-follows graphs) *Given an event log L, its* DFG *is a directed graph (V, E), with the set of vertices V corresponding to the activities in A_L and the set of edges $E \subseteq V \times V$ representing a directly-follows relation $>_L$, defined as: $x >_L y$, if there is a trace $\sigma = \langle e_1, \ldots, e_n \rangle$ and $i \in \{1, \ldots, n-1\}$, such that $\sigma \in L$ and $e_i.a = x$ and $e_{i+1}.a = y$.*

Declarative process models

Unlike imperative process models that explicitly specify allowed process behavior, declarative process models limit behavior through constraints. We describe this paradigm using DECLARE as one of the main declarative languages used in the process mining field [59]. In essence, DECLARE [142] describes a set of constraint templates that can be instantiated to restrict occurrences and orderings of activities in a process. Formally, we define *declarative process models* as follows [59].

Definition 9 (Declarative process models) *A declarative process model DM is a tuple $DM = (Templ, A, K)$. $Templ$ is a finite, non-empty set of constraint templates that are used in DM, where each $templ \in Templ$ is a predicate $\kappa(x_1, \ldots, x_m)$ on variables $x_1, \ldots, x_m$. $A \subseteq \mathcal{A}$ is a finite non-empty set of activities. K is a finite set of constraints, where each $k \in K$ is a tuple $k = (templ, a_1, \ldots, a_m)$, with $templ \in Templ$ and $a_i \in A$ the activity that replaces the variable x_i in $templ$.*

DECLARE templates are essentially abstractions of Linear Temporal Logic (LTL) formulas, which allow to exploit the amenities of LTL verification, with the full complexity of LTL "hidden" from the user. Table 2.4 summarizes common templates and their corresponding LTL formulas [59].

LTL models time as a linear sequence of states, or time points. At each time point, some statements may be true. On this basis, LTL formulas can be used to specify the allowed behavior over the sequence. Using ϕ, ϕ_1, and ϕ_2 to represent LTL formulas, **F**, **X**, **G**, **U**, **S**, and **O** are LTL operators with the following meaning:

- $\mathbf{F}\phi$ means that ϕ holds at some point in the future,
- $\mathbf{X}\phi$ means that ϕ holds in the next instant,
- $\mathbf{G}\phi$ means that ϕ holds forever in the future,
- $\phi_1 \mathbf{U} \phi_2$ means that, at a future point, ϕ_2 will hold and until then ϕ_1 holds,
- $\phi_1 \mathbf{S} \phi_2$ means that, at a point in the past, ϕ_2 held and since then ϕ_1 holds, and
- $\mathbf{O}\phi$ means that ϕ held at some point in the past.

For a formal definition of these operators, we refer to an overview of declarative process modeling and mining [59].

For any declarative constraint, an *activation* of a constraint serves as its triggering condition. Once this condition becomes true, it expects the trace to satisfy the corresponding target condition. Conversely, if the constraint is not activated, the fulfillment of the target condition is not enforced and the constraint is (vacuously) satisfied. For instance, b is an activation condition for PRECEDENCE(a,b) and a is the target condition because the occurrence of b forces a to have occurred before. Table 2.4 specifies the activation condition per template. Given an activity sequence π and a constraint ϕ, assuming that both relate to the same activity set A, π is said to satisfy ϕ, which we denote as $\pi \models \phi$, if ϕ holds in the initial time point of the sequence. We write $\pi \not\models \phi$ if π violates constraint ϕ.

MP-DECLARE extends DECLARE, among others, by introducing additional conditions on attributes of events in a trace σ, i.e., additional activation conditions. To illustrate this concept, let us examine the constraint ABSENCE(*decide on application*), which is always activated at the beginning of a trace, indicating that *decide on application* should never occur. We can use an additional condition on other perspectives that must be satisfied to activate the constraint, though. For instance, we may require that when *decide on application* takes place, it should

Table 2.4: Common DECLARE templates.

DECLARE Template	**LTL formula**	**Activation**
ATLEASTONE(a)	**F** a	at start
ATMOSTONE(a)	$\neg$**F**$(a \wedge$ **X**(**F**$a))$	at start
EXACTLYONE(a)	ATLEASTONE(a)$\wedge$ATMOSTONE(a)	at start
ABSENCE(a)	$\neg$**F** a	at start
RESPONDEDEXISTENCE(a,b)	**F** a $\rightarrow$ **F** b	a
RESPONSE(a,b)	**G**(a $\rightarrow$ **F** b)	a
ALTERNATERESPONSE(a,b)	**G**$(a \rightarrow$ **X**$(\neg a$**U**$b))$	a
PRECEDENCE(a,b)	**G**(b $\rightarrow$ **O** a)	b
ALTERNATEPRECEDENCE(a,b)	$(\neg b$ **S** $a)\wedge$**G** $(b \rightarrow (\neg b$ **S** $a))$	b
COEXISTENCE(a,b)	**F** a $\leftrightarrow$ **F** b	a,b
SUCCESSION(a,b)	RESPONSE(a,b) $\wedge$ PRECEDENCE(a,b)	a,b
ALTERNATESUCCESSION(a,b)	ALTERNATERESPONSE(a,b) $\wedge$ ALTERNATEPRECEDENCE(a,b)	a,b
NOTCOEXISTENCE(a,b)	(**F** a $\rightarrow \neg$**F** b) $\wedge$ (**F** b $\rightarrow \neg$**F** a)	a,b

be performed by a manager and otherwise should not occur. We can express this constraint as ABSENCE(*decide on application*) | `role` $\neq$ *manager*, where `role` is an event attribute and *manager* its required value. Now, if (and only if) the role associated with an event is not *manager*, the ABSENCE constraint is activated.

2.2 Natural Language Processing

Natural Language Processing (NLP) is a research field that investigates the use of computers for processing and understanding human, i.e., natural, languages [54]. In this thesis, we use NLP methods to extract information from natural language elements that are associated with events and process models as well as to assess the similarity between those. For instance, we use NLP methods to extract semantic components from unstructured event attribute names and values using a language model. In this manner, we, e.g., extract the *create* action and the *purchase order* object type from a *purchase order created* activity. To assess the similarity between natural language elements of events and process models we use word representations to identify, e.g., that the two activities *create invoice* and *bill created* are highly similar from a semantic point-of-view, despite strong structural and syntactical differences.

In this section, we describe the main NLP concepts and techniques that are relevant for the remainder of this thesis. When describing them, we aim to clarify their intuition and usefulness and, therefore, put less emphasis on technical aspects. Section 2.2.1 discusses *vector representations*, which allow to capture the meaning of words. Section 2.2.2 introduces *transformer-based language models*, which can be used to tackle a broad range of NLP tasks. Section 2.2.3 introduces so-called *sequence-labeling* tasks that assign labels to individual words of an input sequence, which we frequently use in our approaches. Following the discussion of these general NLP concepts, Section 2.2.4 concludes with a discussion of existing works that apply them in a process analysis context.

2.2.1 Vector Representations of Words

In order to efficiently process data in NLP, words of a given vocabulary are transformed into *vectors*, i.e., numerical representations. As an example, consider the vocabulary $V = \{order, item, package\}$. The simplest way to create vectors for the words in V is to use one-hot encoding, where each word receives one position (*dimension*) in the vector space that is set to 1 in its vector, while all others are set to 0. For V, this would result in the vectors $v_{order} = \langle 1\ 0\ 0 \rangle$, $v_{item} = \langle 0\ 1\ 0 \rangle$, and $v_{package} = \langle 0\ 0\ 1 \rangle$. Vectors created in this manner have a length (*dimen-*

sionality) of $|V|$ and are maximally sparse, i.e., they contain only zeros except for one dimension. This leads to extremely long vectors for large vocabularies. Most importantly, although each word is represented by a distinct vector, these do not capture any additional information about the words, which is often undesirable.

Many NLP tasks require vectors to capture the meaning of the words they represent. The foundation for this is the assumption that words that appear in similar contexts have similar meanings. For instance, because *bill* and *invoice* are used in similar contexts, i.e., they are surrounded by the same words when they appear in texts, they are assumed to be similar. Based on this assumption, vector representations of the meaning of words can be generated from their distributions in natural language text [101].

There are different models that can be used to generate vectors that represent the meaning of words. Early models represent a word as a sparse vector with dimensions corresponding to words in a vocabulary or documents in a collection (similar as in the one-hot example). Each dimension of such a vector is a count of how often a word co-occurs in close proximity to others. More recently, learned, dense vectors—so-called *embeddings*—have been proposed to capture word meaning. It was shown that these outperform sparse vectors in basically every NLP task [101]. Therefore, we use such *embedding* models in our approaches to assess the similarity among natural language components of events and process models.

Embeddings. Embeddings are real-valued vectors that have useful semantic properties. For instance, they capture information on semantic relations such as synonymy between the words they represent. To establish them, *Word2vec* [130] algorithms and variations such as *GloVe* [141] are commonly used.

The intuition of *Word2vec* is that, instead of counting how often each word w occurs in close proximity to a word, e.g., *invoice*, we train a classifier on a binary classification task. In this task, the classifier should answer the question *Is w likely to appear in close proximity to invoice?* The learned classifier's weights can then be used as word embeddings. The advantage is that we can use text as training data for *self-supervised learning*. A word w that occurs in close proximity to the target word *invoice* in a text provides the true label, which can be used to check if the classifier correctly answered the question [101].

GloVe, which stands for Global Vectors, has proven to be especially effective in capturing semantic similarity and, in particular, synonym relations between words [141]. *GloVe* embeddings allow for efficiently computing semantic similarity scores between words and for querying the most similar words in a vocabulary, given a word. Note that we explain how to compute such a score based on two vectors/embeddings below. For instance, querying the five most similar words of *invoice*, yields the words and respective similarity scores listed in Table 2.5. Here,

higher scores indicate higher similarity. As shown, all words are highly related from a semantic point of view. Notably, also terms that are syntactically completely unrelated to *invoice*, e.g., *payment* and *billing*, are considered highly similar.

Word	**Similarity score**
invoices	0.756
receipt	0.643
invoicing	0.636
payment	0.624
billing	0.604

Table 2.5: The most similar words of *invoice* computed using *GloVe* embeddings.

GloVe embeddings are trained in an unsupervised manner on global word-word-co-occurrence statistics of an entire text corpus, i.e., on statistics on how often word pairs occur near each another in a corpus [141]. There are different corpora used as a basis for this training, such as the entire English Wikipedia. In essence, individual words in the corpus are mapped into a meaningful vector space so that the distance between the vectors that represent the words is related to semantic similarity between them [101].

Because of their capability of capturing semantic similarity in a precise manner, we use *GloVe* embeddings in part of our semantic-annotation approach that we describe in Chapter 3. Among others, we use them to identify the type of actions, e.g., whether an action refers to an update of an application or a decision about it.

Vector similarity. As outlined above, a key feature of vector representations is that they allow for assessing similarities between the words they represent.

Such similarities are computed through a function of the dot product between vectors of equal length n, as defined in Equation 2.1. The most common function is the *cosine similarity* of two vectors v and w, as defined in Equation 2.2, which we also use when computing the semantic similarity between words in our approaches.

$$v \cdot w = \sum_{i=1}^{n} v_i w_i \tag{2.1}$$

$$cosine(v, w) = \frac{v \cdot w}{|v||w|} \tag{2.2}$$

2.2.2 Transformer-Based Language Models

Transformer-based language models are machine learning models that are capable of efficiently learning dependencies in natural language text. They are trained to

develop a general understanding of a language and can then be specialized for a wide range of NLP tasks. Their remarkable performance across tasks can primarily be attributed to the following key features:

Attention. Transformer-based language models, such as the well-known *Bidirectional Encoder Representations from Transformers* (BERT) model [56], rely on the *(self-)attention mechanism* [178]. This mechanism allows them to selectively pay attention to words in the surroundings of a given word, which are relevant for the meaning of that word. For example, in the sentence *She runs a company*, the attention mechanism considers the word *company* while processing the word *runs*. This is essential for inferring the meaning of *runs* in the given context, which is *to manage*. As such, attention allows transformer-based language models to generate *contextualized embeddings*. In contrast to the *static embeddings* discussed previously, contextualized embeddings exhibit contextual awareness. In particular, unlike *Word2Vec* and *GloVe*, which generate a single static embedding for each distinct word in a corpus, BERT tailors its embeddings to the specific context in which a word appears [101]. For example, given the two sentences *She runs a company* and *She runs a marathon*, BERT learns two different embeddings for *runs*, due to the different meanings in the respective contexts.

Pretraining. Transformer-based language models are typically *pretrained*, which means that they are fed with considerably large amounts of text in order to learn a model of the meaning of words or even full sentences [161]. As for *Word2Vec*, this is typically done via self-supervised learning. Unlike supervised learning, which requires large amounts of labeled data for a specific task, self-supervised learning leverages large amounts of unlabeled data (huge text corpora) for a certain pretraining objective. In case of BERT, this objective is *masked-language modeling*. A certain amount of tokens, i.e., (parts of) words, in each training sequence is replaced by a special *mask* token. The goal is then to "predict" the true tokens that were masked. This objective is particularly suitable for bidirectional training (employed by BERT), where the context (the surrounding tokens) both on the left and the right side of a masked token are considered to make a prediction [56]. This allows BERT to take the entire context of an input sequence into account.

Fine-tuning. Fine-tuning is the process of further training a pretrained model. The goal is to specialize the model for a specific task, such as named-entity recognition or semantic role labeling (that we introduce below). The training data size for fine-tuning is considerably smaller than for pretraining. In this manner, fine-tuning pretrained models reduces the effort to create labelled training data and the resources needed to train a model for a specific task from scratch [101].

We use BERT in our semantic-annotation approach that we describe in Chapter 3. In particular, we fine-tune it to label parts of activity labels with semantic roles such as *action*, *object*, and *actor*. Furthermore, we use contextualized embeddings

of a pretrained transformer in our approach for detecting best-practice violations in event data that we describe in Chapter 7. In particular, we use these to assess the similarity between natural language components of declarative constraints and events. These commonly consist of multiple words, e.g., *purchase order requisition*, so that we benefit from the context-awareness that these embeddings provide.

2.2.3 Sequence Labeling

Sequence labeling tasks aim to assign a label to each word in an input word sequence. Such tasks can be satisfied using classic techniques such as Hidden Markov Models (HMM) and Conditional Random Fields (CRF). However, with the recent advancements in NLP, state-of-the-art sequence-labeling techniques employ Transformers that consistently outperform classic techniques on these tasks [101].

Prominent examples of sequence labeling tasks are part-of-speech tagging, named-entity recognition, and semantic role labeling, which we also use in our approaches and briefly introduce next.

Table 2.6: Universal POS tag set.

Label	Description	Examples	Label	Description	Examples
ADJ	adjective	*old*, *green*	DET	determiner	*this*, *which*
ADV	adverb	*briefly*	NUM	numeral	*one*, *2*
INTJ	interjection	*psst*, *ouch*	PART	particle	*not*, *'s*
NOUN	noun	*item*	PRON	pronoun	*you*, *someone*
PROPN	proper noun	*Microsoft*	SCONJ	subordinating conjunction	*if*, *while*
VERB	verb	*create*			
ADP	adposition	*in*, *during*	PUNCT	punctuation	*.,,,()*
AUX	auxiliary	*has*, *will*	SYM	symbol	*$, %*
CCONJ	coordinating conjunction	*and*, *or*	X	other	*xfgh*, *kfql*

Part-of-speech (POS) tagging. In POS-tagging, we aim to assign linguistic roles to the words of the input sequence. There are various label sets that can be used for POS tagging such as the *Universal POS tag set*[1]. The individual tags, their meaning, and examples of that tag set is depicted in Table 2.6. Using it as a basis, a POS-tagger labels the word sequence *the vendor creates an invoice in SAP* as *the*\DET, *vendor*\NOUN, *creates*\VERB, *an*\DET, *invoice* \NOUN, *in*\ADP, *SAP*\PROPN.

We use POS-tagging as part of our semantic-annotation approach, which we describe in Chapter 3, and our approach for recognizing task-level events, which we describe in Chapter 5.

[1] https://universaldependencies.org/u/pos/

Named-entity recognition (NER). In NER, we aim to find spans of words that correspond to proper names and assign them an entity type. As for POS tagging, there are various label sets that can be used as a basis. One such set is the *CoNLL 2003 NER tag set* [166], which defines the PER (person), LOC (location), ORG (organization), and MISC (miscellaneous) labels. Given an input sequence *Pete Miller works for Microsoft*, a NER-tagger that uses this set would return *[Pete Miller]*$_{\text{PER}}$ *works for [Microsoft]*$_{\text{ORG}}$.

We use NER in our semantic-annotation approach described in Chapter 3 and our task-recognition approach described in Chapter 5.

Semantic-role labeling (SRL). In SRL, we aim to assign semantic roles to spans of words. The task's goal is to answer questions like *Who is doing what, where and to whom?* Various standard but also custom labeling sets can be used for SRL, such as the so-called *PropBank* [103] labeling set. For instance, for the input sequence *Pete Miller works for Microsoft*, an SRL technique that uses *PropBank* would return *[Pete Miller]*$_{\text{ARG0}}$ *[works]*$_{\text{PRED}}$ *for [Microsoft]*$_{\text{ARG2}}$. In particular, *PropBank* defines semantic roles with respect to an individual verb sense. In our example, ARG0 thus translates into *worker* and ARG2 into *beneficiary*.

In this thesis, we label activities with semantic components as part of our semantic-annotation approach (Chapter 3), which represents an SRL task.

2.2.4 Natural Language Processing for Process Analysis

There is a range of works that apply NLP methods for the purpose of (automated) process analysis. This section provides an overview of such works, distinguishing between ones that mainly focus on the analysis of event data, process models, and textual process descriptions.

Analysis of event data. The analysis of event data using methods from NLP primarily focuses on the activity labels that are associated with events. Research on analyzing such labels includes the work by Deokar and Tao [55], which groups semantically similar labels together in order to reduce the complexity of an event log. Ramos-Gutiérrez et al. [147] aim to improve event log quality by relabeling activities based on proposed quality metrics. To this end, they suggest standard NLP tasks, such as POS-tagging, based on the computed quality of an event log. In order to detect process behavior in event logs, which is problematic from a semantic point of view, e.g., cases where items are shipped after an order has been cancelled, the notion of *semantic anomaly detection* has recently been proposed [5]. Existing approaches detect such behavior through rules that they extract from linguistic resources and process models [5] or by training a classifier to identify problematic ordering of activity pairs [43].

Analysis of process models. Research on the analysis of process models with NLP methods focuses, among others, on the categorization of process activities, achieved by mapping process model components to existing taxonomies [114] and on categorizing activities according to their degree of automation [111]. Other work focuses on the automated generation of process model names [112] or activity labels for groups of lower-level process steps [113]. Furthermore, label parsing [110] is used to analyze the labeling style of activities and to extract the actions and objects from these, which corresponds to an SRL task that we outlined above.

Analysis of textual process descriptions. Research on the analysis of textual process descriptions focuses on extracting process models from such descriptions [1], comparing process models to them [4], and checking conformance between event logs and textual process descriptions [3]. Barrientos et al. [28] address the problem of automatically checking if event log traces comply with natural language regulatory documents. Winter et al. [182] focus on automatically assessing the compliance of process models and regulatory documents instead, whereas Sai et al. [165] aim to detect deviations between external regulatory requirements and manually derived internal ones. With the advent of *large language models*, their applicability for such tasks, e.g., for process model extraction from text, has been investigated. Initial results indicate that similar or better performance as state-of-the-art approaches can be achieved for such tasks [87].

Chapter 3

Semantic Annotation of Event Logs

Organizations frequently seek insights into the execution of their processes beyond the control-flow. Certain process mining techniques can provide such insights by taking into account the meaning of events or their attributes. For instance, social network analysis [12] considers the actors that perform events, object-centric process analysis [6] considers the objects that are handled in a process, and semantic anomaly detection [5] detects undesired process behavior through the meaning of performed actions.

However, the information required for applying these techniques, such as actions, objects, and actors associated with events, is not readily available in most event logs. A prime cause for this is the limited standardization of event data, which is neither enforced nor complete with respect to the relevant pieces of information, which we call *semantic components*. As a consequence, these components are often contained in unstructured textual attributes or in attributes that are unclearly named and, thus, cannot be used by process mining techniques.

In this chapter, we propose an approach that alleviates this problem by automatically identifying semantic components in events. In particular, our approach aims to identify information on eight semantic component types, covering various kinds of information related to objects, actions, actors, and other resources. After this, it further categorizes the identified actions and actors into various categories, allowing for a more detailed analysis of the way in which a process is executed and what kind of resources are involved in its execution. To achieve its goal, our approach combines a transformer-based language model, fine-tuned for SRL, with novel techniques for semantic attribute classification and component categorization. We assess the accuracy of our approach by applying it on a collection of 14

real-world event logs. Furthermore, we demonstrate its usefulness by showcasing three application cases that are only possible thanks to the semantic annotations that our approach provides. Specifically, we show how our approach can be used to refine event labels in order to reduce the complexity of discovered process models, enable object-centric process analysis, and analyze the automation degree of processes based on actor information.

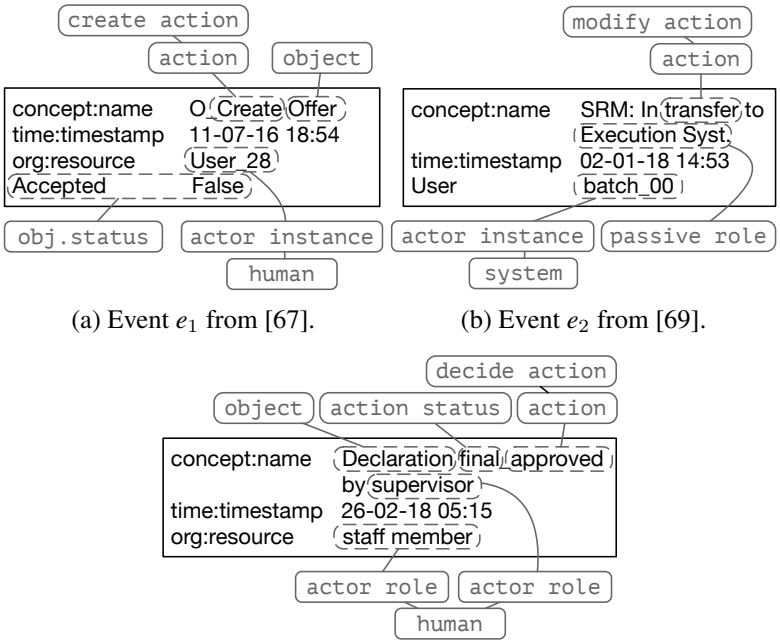
This chapter is based on a paper titled "*Enabling semantics-aware process mining through the automatic annotation of event logs*" [148] by Adrian Rebmann and Han van der Aa, which itself is an extension of the conference paper "*Extracting semantic process information from the natural language in event logs*" [149].

The remainder of the chapter is structured as follows. Section 3.1 illustrates the problem of annotating events with semantic components using examples from real-world event logs. Section 3.2 defines the scope of our work in terms of the covered semantic information and the main aspects involved in the annotation task. Section 3.3 presents our annotation approach. Section 3.4 reports on evaluation experiments that show that our approach achieves accurate results on real-world event logs, spanning various domains and varying considerably in terms of their informational structure. As part of the evaluation, we also highlight the usefulness of our approach by using it in three application cases. Section 3.5 reflects on limitations of our work. Finally, Section 3.6 discusses streams of related work, whereas Section 3.7 summarizes the results.

3.1 Problem Illustration

In this section, we illustrate the problem of annotating events with semantic components. Figure 3.1 shows three events from real-world event logs, capturing information on semantic components in various manners.

All of these example events refer to an activity (`concept:name`) and a timestamp (`time:timestamp`), specified using attributes from the *eXtensible Event Stream* (XES) standard [88], whereas event e_1 also uses the standard `org:resource` attribute to capture which actor performed the event. However, this XES standard is not always followed properly. For example, event e_2 uses `User`, rather than the standard `org:resource` attribute, to indicate the actor, whereas event e_3 erroneously uses this standard attribute to capture information on the actor's role (a *staff member*) rather than on the specific actor instance. Furthermore, the XES standard only covers a limited set of attributes, which means that information on semantic components such as *actions*, *object types*, and their *status* are not covered by the standard at all. Particularly problematic here is that information on these and other components is commonly not explicitly captured through event at-



(a) Event e_1 from [67].

(b) Event e_2 from [69].

(c) Event e_3 from [70].

Figure 3.1: Exemplary events and their semantic components.

tributes, but is rather part of unstructured, textual attributes associated with events, usually as part of their activities. For example, the *Declaration final_approved by supervisor* activities of e_3 captures the event's object (*declaration*), the action (*submitted*) along with its status (*final*), and the role of the actor (*supervisor*).

Since these components are all contained in the same attribute value, the information cannot be used by process mining techniques. Enabling this use requires the processing of each individual attribute value in order to identify the included semantic information. Clearly, this is an extremely tedious and time-consuming task when considered in light of the complexity of real-world logs, with hundreds of event classes, dozens of attributes, and thousands of instances. Therefore, this calls for automated support for the semantic annotation of event data, to make relevant semantic components available to process mining techniques.

3.2 Scope

This section describes the scope of our approach in terms of the semantic information it covers and the main aspects involved in the annotation task. In the following, Section 3.2.1 covers the scope with respect to the identification of semantic

components in event data, whereas Section 3.2.2 discusses the subsequent further categorization of identified components into more specific types.

3.2.1 Semantic Component Identification

Given an event log L, our work first annotates pieces of information associated with events that correspond to particular *semantic components*. This identification task covers the following semantic component types and aspects.

Semantic Component Types

Our work covers various component types that support a detailed analysis of process execution from a behavioral perspective, i.e., we target semantic components that are commonly observed in event logs and that are relevant for an order-based analysis of event data. Therefore, we consider information related to *objects*, *actions*, and active and passive *resources* involved in a process' execution. For each of these categories, we define multiple semantic component types:

Objects. We use the term *object* to refer to any artifact or entity that is being handled in a process, which covers documents such as an *offer* or a *declaration*, physical objects such as a *car* or a *computer*, but can also relate to, e.g., a *customer* or an *applicant*. For the events in a log, our work annotates two component types:

- `object` as the type of a object, such as an *offer* or a *declaration* associated with an event, and
- $\texttt{object}_{status}$ as the reported status of an object, e.g., whether an identified order is indicated as being *open*, *canceled*, or *accepted*.

Actions. Similarly, we define two types of components to capture information on the actions that are applied to objects:

- `action` as the performed action itself, e.g., *create*, *transfer*, or *approve*, and
- $\texttt{action}_{status}$ as further information on its status, e.g., *started*, *paused*, *final*, or *completed*.

Resources. We capture information regarding the active resources of events, i.e., the entities that performed the recorded actions, with two component types:

- $\texttt{actor}_{role}$ as the type or role of active resource in the event, e.g., a *supervisor* or a *system*, and
- $\texttt{actor}_{instance}$ for information indicating the specific actor instance, e.g., an employee identifier such as *User_28*.

Aside from the actor, events may capture *passive* resources involved in an event, primarily in the form of *recipients*. For this, we again define two component types:

- `passive`$_{role}$ as the type or role of passive resource related to the event, e.g., the *supervisor* receiving a document or a *system* on which a file is stored or transferred through, and
- `passive`$_{instance}$ for information indicating the specific resource, e.g., an employee or system identifier such as *batch_00*.

Coverage and extensibility. The semantic component types considered here enable a broad range of insights into the execution of a process. For example, the *object* and *action* categories allow one to obtain detailed insights into the objects moving through a process, their inter-relations, and their life-cycles. Furthermore, by also considering the resource-related components, one can, for instance, gain detailed insights into the resource behavior associated with a particular object, e.g., how resources jointly collaborate on the processing of a specific document.

While the covered component types were purposefully selected based on their relevance in real-world event logs, our approach is not limited to these specific component types. Given that we employ state-of-the-art NLP methods and classification models that generalize well, the availability of appropriate event data allows our approach to be easily extended to cover additional semantic component types, both within and outside the informational categories considered here.

The Component Identification Task

In order to make sure that all relevant information is identified in an event log, our work considers two aspects of the *semantic component identification* task, concerned with two kinds of event attributes: *attribute-level classification* for attributes dedicated to a single semantic component type and *instance-level labeling* for textual attributes covering various component types:

Attribute-level classification. Attribute-level classification determines the component types of attributes that provide the same kind of information throughout an event log, e.g., a `doctype` attribute indicating an object (`object`) or an `org:resource` attribute capturing information on the specific resource performing an event (`actor`$_{instance}$).

Although the XES standard definition [88] specifies several dedicated event attributes, such as `org:resource` and `org:role`, these only cover a subset of the semantic component types relevant for our approach. In particular, they omit component types related to objects, actions, and passive resources. Information on these types may, thus, be captured in attributes with diverse names, e.g., in the real-world *hospital log* [123], the status of objects (`object`$_{status}$) is jointly indicated by several event attributes, such as `isCancelled` and `isClosed`. Furthermore, even for component types covered by standard attributes, there is no guarantee that event logs adhere to the conventions, e.g., rather than using `org:group`, the *Business Pro-*

cess Intelligence (BPI) event log from 2014 [65] captures information on actors in an `Assignment_Group` attribute.

Instance-level labeling. Instance-level labeling, in turn, identifies semantic information for attributes with unstructured, textual values that encompass various semantic component types, differing per event. This task is most relevant for activity labels that are (in case of XES) stored in a `concept:name` attribute.

Log	ID	Activity label	Semantic components
WABO [38]	l_1	T08 Draft and send request for advice	`action` ($\times 2$), `object`
BPI15 [66]	l_2	send design decision to stakeholders	`action`, `object`, `passive`$_{role}$
BPI15 [66]	l_3	send letter in progress	`action`, `object`, `action`$_{status}$
RTFM [53]	l_4	Insert Date Appeal to Prefecture	`action`, `object`, `passive`$_{role}$
BPI19 [69]	l_5	Vendor creates invoice	`actor`$_{role}$, `action`, `object`
BPI19 [69]	l_6	SRM: In transfer to Execution Syst.	`action`, `passive`$_{role}$
BPI20 [70]	l_7	Declaration final_approved by supervisor	`object`, `action`$_{status}$, `action`, `actor`$_{role}$
BPI17 [67]	l_8	O_Create Offer	`action`, `object`

Table 3.1: Activity labels from real-world logs with their semantic components.

Activity labels contain highly valuable semantic information, but also present considerable challenges to their proper handling, as illustrated through the real-world activities in Table 3.1. The examples highlight the diversity of natural language labels, in terms of their structure and the semantic component types that they cover. It is worth mentioning that such differences may even exist for labels within the same event log, e.g., labels l_5 and l_6 (the label of e_2 of the running example) differ considerably in their textual structure and the information they cover, yet they both stem from the same event log. Another characteristic to point out is the possibility of recurring component types within a label, such as seen for label l_1, which contains two `action` components: *draft* and *send*. Thus, an approach for instance-level labeling needs to be able to deal with textual attribute values that are highly variable in terms of the information they convey, as well as their structure.

3.2.2 Semantic Component Categorization

Once semantic components have been identified, we apply a component categorization step. In this step, we use *action categorization* to classify identified actions

into predefined types of high-level actions, whereas we use *resource categorization* to distinguish between human and system actors involved in a process.

Action categorization. Because of the wide range of domains in which organizational processes occur, processes can consist of a plethora of different actions, resulting in a virtually unlimited universe of potential actions. Thanks to the state-of-the-art NLP technology on which our work builds, our component identification approach can still recognize `action` components in an accurate manner, despite this variety. Nevertheless, various kinds of process analysis, such as abstraction, filtering, and conformance checking, can benefit from having an understanding about which actions (or activities) in a process serve a similar purpose and what that purpose actually entails, e.g., *improving* and *updating* both refer to a modification of a certain object. This calls for a reduction of the range of actions observed in a process to a set of known higher-level action types.

To operationalize this, we use *action categorization* to assign an action type to each `action` component identified for a process. As a basis for this categorization, we adopt the established action classification framework of the MIT Process Handbook [122], which classifies process-related actions in a hierarchical manner. The top-most level of this hierarchy defines eight high-level actions, as follows:

- *Create*: An action is classified as *create* if its essence is focused on the creation of an output of some sort, e.g., *producing* or *documenting* something.
- *Modify*: An action is classified as *modify* if its essence is focused on changing some attribute of the input as the output, e.g., *improving* or *sending* something.
- *Preserve*: An action is classified as *preserve* if its essence is to keep the input unchanged as an output, just at a later point in time, e.g., *storing* or *packaging* something.
- *Destroy*: An action is classified as *destroy* if its essence is focused on the destruction of an input of some sort, e.g., *retiring* or *eliminating* something.
- *Combine*: An action is classified as *combine* if its essence is grouping or integrating multiples of an input into a single collected output, e.g., *grouping* or *matching* something.
- *Separate*: An action is classified as *separate* if its essence is ungrouping or splitting a single collected input into multiple outputs, e.g., *dividing* or *extracting* something.
- *Decide*: An action is classified as *decide* if its essence is a choice among multiple alternatives, e.g., *determining* or *approving* something.
- *Manage*: An action is classified as *manage* when the actual process to be used is yet unspecified, such as a means of coordinating a dependency or other process, e.g., *assigning* or *scheduling* something.

Categorizing actions into these top-level action types then enables a more detailed view on the process allowing analyses based on their meaning. For instance, it allows us to separate a process into different stages, such as creation, processing (modifying), and decision phases (see Section 4.4.4 for an application case that shows this). Note that our work is independent of this specific classification scheme. It can be replaced with any categorization for which instance data is available, such as the broader verb classification framework by Levin [115].

Resource categorization. Knowing whether process steps are performed by humans or systems allows for a detailed analysis of a process with respect to the resource perspective. For instance, it enables an assessment of the degree of automation or system support of a process (cf. Section 3.4.4). Furthermore, in the context of organizational mining, the interactions among employees are of particular interest, which requires the ability to distinguish human from non-human resources. To enable such analyses, we further categorize identified actors into system and human resources through *resource categorization*.

Although this categorization thus involves only two classes, properly operationalizing it is particularly interesting. Specifically, a categorization approach has to take into account that the nature of a resource can be derived from various kinds of information, which may or may not be available in a given event log or for a particular resource. For example, while $\texttt{actor}_{role}$ components that indicate roles such as a *supervisor* or a *service*, already reveal the category of an actor from a semantic perspective, such clear descriptions are rarely available. Similarly, $\texttt{actor}_{instance}$ components can be expressive, e.g., *User_28* or *batch_00*, but may also be unspecific, e.g., *res_90*. Thus, both types of semantic components can provide useful information, but cannot be solely relied on. Nevertheless, even when no meaningful information is available in the actor-specific components themselves, insights about the resource category may still be derived by looking at the context in which an actor operates. This can, for example, be achieved by considering the kind of activities an actor performs—a resource associated with *SRM: In transfer to Execution Syst.* events is likely to be a system—whereas also the duration of activities (instant versus highly variable), can be useful to distinguish between actor types.

3.3 Annotation Approach

This section describes our approach for the semantic annotation of event logs. Section 3.3.1 first introduces the approach at a high level, whereas Section 3.3.2–Section 3.3.4 describe its main stages in detail. Finally, Section 3.3.5 describes the output our approach generates.

3.3.1 Approach Overview

Our approach takes as input an event log L according to Definition 3 and annotates the events in E_L with additional information about the semantic components contained in an event's attributes[1], as described in Section 3.2. To achieve this, our approach consists of three main parts, as illustrated in Figure 3.2.

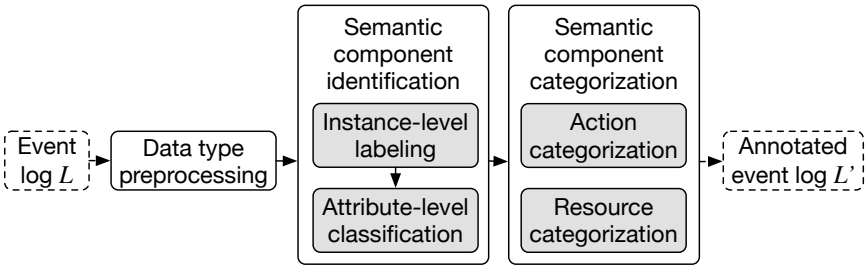


Figure 3.2: Overview of the annotation approach.

Given an event log L, the *data type preprocessing* step first analyzes the domains of attributes in $\mathcal{D}_L$ in order to differentiate among textual, miscellaneous, and irrelevant attributes. Next, the *semantic component identification* stage aims to annotate each event with up to eight different component types. This stage consists of two subsequent steps: *instance-level labeling*, which annotates the individual values of textual attributes, and *attribute-level classification*, which determines the appropriate component type for entire attributes. Afterwards, our approach proceeds with the *semantic component categorization* stage, which consists of two independent steps: in the *action categorization* step, identified `action` components are classified according to eight high-level types, whereas *resource categorization* differentiates between `human` and `system` resources. As output, our approach returns an augmented event log L', in the XES format, in which each event e is extended with additional semantic information, i.e., with its action(s) and action type(s), object(s), and various kinds of resource-related information.

3.3.2 Data Type Preprocessing

In its first step, our approach identifies a set of textual attributes $\mathcal{D}_L^T$, serving as input for the instance-level labeling step, and a set of miscellaneous attributes $\mathcal{D}_L^M$, serving as input to the attribute-level classification step. Attributes that are not

[1]Note that we do not impose any assumptions on the attributes contained in $\mathcal{D}_L$, meaning that we do not assume that standard attributes such as `org:role` are included in $\mathcal{D}_L$.

included in either of these sets are deemed irrelevant for our purposes and, thus, omitted from further consideration. This preprocessing step works as follows.

Data type classification. We first use standard techniques, such as provided by the *Pandas* library[2], to classify each attribute in $\mathcal{D}_L$ as either `timestamp`, `numeric`, `boolean`, or `string`, based on its domain.

Identifying textual attributes. In order to identify the set of textual attributes $\mathcal{D}_L^T$, we differentiate between `string` attributes with true natural language values, e.g., *Declaration final_approved by supervisor* or *O_Create Offer*, and other kinds of alphanumeric attributes, with values such as *User_28*, *A*, and *R_45_2A*. Only the former kind of attributes will be assigned to $\mathcal{D}_L^T$ and, thus, analyzed on an instance-level in the remainder. We identify such textual attributes as follows:

1. Given a `string` attribute, we first apply a tokenization function `tok`, which splits an attribute value into lowercase tokens (based on whitespace, camel-case, underscores, etc.) and omits any numeric ones. E.g., given s_1 = *Declaration final_approved by supervisor*, s_2 = *User_28*, and s_3 = *08_AWB45_005*, we obtain: `tok`(s_1) = [*declaration*, *final*, *approved*, *by*, *supervisor*], `tok`(s_2) = [*user*] and `tok`(s_3) = [*awb*].
2. We apply a POS-tagger, provided by standard NLP tools such as *spaCy* [97], to assign each token a tag from the Universal Part of Speech tag set[3]. Thus, we obtain [(declaration, NOUN), (final, NOUN), (approved, VERB), (by, ADP), (supervisor, NOUN)] for s_1, [(user, NOUN)] for s_2, and [(awb, PROPN)] for s_3.
3. Finally, we exclude any attribute that only has values with the same token in `tok`(s) or that do not contain any NOUN, VERB, ADV, ADP, or ADJ tokens. In this way, we omit attributes with values such as s_2 = *User_28* and s_3 = *08_AWB45_005*, which are identifiers, rather than real textual attributes. The other attributes, which have diverse, textual values, e.g., s_1 = *Declaration final_approved by supervisor*, are assigned to $\mathcal{D}_L^T$.

Selecting miscellaneous attributes. We also identify a set of non-textual attributes that are candidates for semantic labeling, referred to as the set of miscellaneous attributes, $\mathcal{D}_L^M \subseteq \mathcal{D}_L \setminus \mathcal{D}_L^T$. This set contains attributes that are not included in $\mathcal{D}_L^T$, yet have a data type that may still correspond to certain semantic component types, such as statuses or identifiers.

In order to establish $\mathcal{D}_L^M$, we first discard those attributes in $\mathcal{D}_L \setminus \mathcal{D}_L^T$ categorized as `timestamp` attributes, as well as `numeric` attributes that include *real* or *negative* values. We exclude these because they are not used to capture semantic information. By contrast, the remaining attributes have data types that may corre-

[2] `https://pandas.pydata.org`

[3] `https://universaldependencies.org/docs/u/pos/`

spond to component types, such as Boolean attributes that can be used to indicate specific states, e.g., `Accepted`, whereas non-negative integers are commonly used as identifiers, e.g., a `customer` attribute with values such as *32015* and *49102*. These remaining attributes are then joined with the `string` attributes that were not selected for $\mathcal{D}_L^T$, e.g., attributes with values such as the aforementioned *User_28*, and *08_AWB45_005* examples, to form the set of miscellaneous attributes $\mathcal{D}_L^M$.

In this manner, given the log of example event e_1, e.g., `concept:name` and `org:resource` are assigned to $\mathcal{D}_L^T$, `Accepted` to $\mathcal{D}_L^M$, and `time:timestamp` is omitted from consideration.

3.3.3 Semantic Component Identification

The *semantic component identification* stage comprises two subsequent steps. First, *instance-level labeling* processes the values of textual attributes to extract the parts that correspond to semantic component types, e.g., recognizing that a *document received* activity label contains the object *document* and the action *received*. Afterwards, the *attribute-level classification* step identifies the appropriate component type for each of the remaining attributes, i.e., it aims to determine the semantic component type that corresponds to all values of a certain attribute by considering its value domain as well as its name. For example, it recognizes that all values of a `doctype` attribute correspond to the `object` component type. The details of these steps are as follows.

Instance-level Labeling of Textual Attributes

In this step, our approach annotates the values of textual attributes in order to extract the parts that correspond to certain semantic component types, e.g., recognizing that the *O_Create Offer* label of event e_1 contains *offer* as the `object` and *create* as the `action`. As discussed in Section 3.2.1, this task comes with considerable challenges due to the high diversity of textual attribute values in terms of their linguistic structure and informational content. To be able to deal with these challenges, we therefore build on state-of-the-art NLP methods.

Semantic-role-labeling task. We consider the labeling of textual attribute values with semantic component types as an SRL task. Therefore, we instantiate a function that assigns one of the eight semantic component types (i.e., semantic roles) described in Section 3.2.1 to chunks (i.e., groups) of consecutive tokens from a tokenized textual attribute value. Formally, we use $\mathcal{T}$ to refer to the set of component types and we denote the outcome of the tokenization of an attribute value for a given event $e \in E_L$ and attribute $d \in \mathcal{D}_L^T$, as $\texttt{tok}(e.d) = \langle w_1, \ldots, w_n \rangle$, where each token represents a word w_i. Then, we define a function $\texttt{tag}(\langle w_1, \ldots, w_n \rangle) \rightarrow$

$\langle c_1 \backslash t_1, \ldots, c_m \backslash t_m \rangle$, where c_i for $1 \leq i \leq m$ is a chunk consisting of one or more consecutive tokens from $\langle w_1, \ldots, w_n \rangle$, with $t_i \in \mathcal{T} \cup \{\texttt{other}\}$ as its associated semantic component type (or `other` for words that do not correspond to any targeted semantic component). For example, $\texttt{tag}(\langle \textit{create}, \textit{offer} \rangle)$ results in $\langle \textit{create} \backslash \texttt{action}, \textit{offer} \backslash \texttt{object} \rangle$ and $\texttt{tag}(\langle \textit{declaration}, \textit{final}, \textit{approved}, \textit{by}, \textit{supervisor} \rangle)$ yields $\langle \textit{declaration} \backslash \texttt{object}, \textit{final} \backslash \texttt{action}_{status}, \textit{approved} \backslash \texttt{action}, \textit{by} \backslash \texttt{other}, \textit{supervisor} \backslash \texttt{actor}_{role} \rangle$.

Fine-tuning BERT. To instantiate the `tag` function, we employ BERT [56] (cf. Section 2.2). In particular, we fine-tune a pretrained BERT model on the task of tagging chunks of textual attribute values that correspond to semantic component types. To this end, we manually labeled a collection of 13,231 unique textual values stemming from existing collections of process models [110], textual process descriptions [111], and event logs (see Section 3.4.1). As expected, the collected samples do not capture information on resource instances, and rather contain information on the type level (i.e., $\texttt{actor}_{role}$ and $\texttt{passive}_{role}$). For those semantic component types that are included in the samples, we observe a considerable imbalance in their commonality, as depicted in Table 3.2. In particular, while component types such as `object` (14,629 times), `action` (12,573), and even $\texttt{passive}_{role}$ (1,191) are relatively common, we only found few occurrences of $\texttt{actor}_{role}$ (135), $\texttt{object}_{status}$ (92), and $\texttt{action}_{status}$ (30) component types.

Table 3.2: Characteristics of the training data used to fine-tune BERT ($s = status$).

Source	**Count**	**object**	**object**$_s$	**action**	**action**$_s$	**actor**$_{role}$	**passive**$_{role}$	**other**
Models	11,658	13,543	50	11,445	3	58	1,058	4,966
Desc.	498	503	11	498	0	8	114	206
Logs	625	583	31	630	27	69	19	291
Aug.	450	350	100	350	150	200	0	150
Total	13,231	14,979	192	12,923	180	335	1,191	5,613

To address this imbalance, we created additional samples with $\texttt{object}_{status}$, $\texttt{action}_{status}$, and $\texttt{actor}_{role}$ component types using established data augmentation strategies. In particular, we created samples by complementing randomly selected textual values with (1) known $\texttt{actor}_{role}$ descriptions, e.g., *offer created* is extended to *offer created by supervisor*, and (2) common life-cycle transitions [7] to create samples containing $\texttt{object}_{status}$ and $\texttt{action}_{status}$ component types, e.g., *check invoice* is extended to *check invoice completed*. However, as shown in Table 3.2, we limited the number of extra samples to avoid overemphasizing the importance of these component types.

Given this training data, we operationalize the tag function using the ***BERT base uncased pre-trained language model***[4] with 12 transformer layers, a hidden state size of 768 and 12 self-attention heads. As suggested by its developers [56], we trained 2 epochs using a batch size of 16 and a learning rate of 5e-5.

Reassigning noun-only attributes. Having applied the tag function to the values of an attribute $d \in \mathcal{D}_L^T$, we check whether the tagging is likely to have been successful. In particular, we recognize that it is hard for an automated technique to distinguish among the `object`, $\texttt{actor}_{role}$, and $\texttt{passive}_{role}$ component types, when there is no contextual information, since their values all correspond to nouns. For instance, the `User` attribute of event e_2, which encompasses noun-based values like *user* and *batch*, may be falsely tagged as `object` rather than $\texttt{actor}_{role}$. This happens because objects are much more common in the training data and the attribute values do not provide any further context that indicates the correct component type. To overcome this issue, we establish a set $\mathcal{D}_L^N \subseteq \mathcal{D}_L^T$ that contains all such *noun-only* attributes, i.e., attributes of which all values correspond solely to the `object` component type. This set is then forwarded to the attribute-level classification step of our approach, whereas the tagged values of the other attributes directly become part of our approach's output in the form of semantic annotations.

In this manner, we annotate the values of our example events' `concept:name` attributes. For e_1: *create*: `action`, *offer*: `object`, for e_2: *transfer*: `action`, *execution sys.*: $\texttt{passive}_{role}$, and e_3: *declaration*: `object`, *final*: $\texttt{action}_{status}$, *approve*: `action`, *supervisor*: $\texttt{actor}_{role}$. The events' noun-only textual attributes, i.e., `org:resource` and `User`, are reassigned to be handled in the next step.

Attribute-Level Classification

This step determines the semantic component types of the miscellaneous attributes in $\mathcal{D}_L^M$, such as the Boolean `Accepted` attribute of e_1, identified in the preprocessing stage, and the noun-only textual attributes in $\mathcal{D}_L^N$, such as `User` of e_2 and `org:resource` of e_3, stemming from the previous step. We target this task as a classification problem at the attribute level, i.e., we aim to identify a single semantic component type $t \in \mathcal{T} \cup \{\texttt{other}\}$ for each $d \in \mathcal{D}_L^M \cup \mathcal{D}_L^N$ and then assign type t to each occurrence of d in the event log. For attributes in $\mathcal{D}_L^M$, our approach operationalizes this classification task based on just an attribute's name, whereas it considers the name as well as the values of attributes in $\mathcal{D}_L^N$.

Note that we initially assign each attribute a component type $t \in \mathcal{T}'$, where $\mathcal{T}'$ excludes the *instance* component types, i.e. $\texttt{actor}_{instance}$ and $\texttt{passive}_{instance}$,

[4] `https://github.com/google-research/bert`

from $\mathcal{T}$. Afterwards, our approach distinguishes between type-level and instance-level resource attributes based on their domain.

Classifying miscellaneous attributes in $\mathcal{D}_L^M$. In order to determine the component type of miscellaneous attributes, we recognize that their values, typically alphanumeric identifiers, integers or Boolean, are mostly uninformative and thus not helpful for the classification task. Therefore, we determine the component type of an attribute $d \in \mathcal{D}_L^M$ based on its name. To do this, we build a classifier that classifies a $\texttt{name}(d)$ based on a set of available attribute names, which we each manually assigned a class from the set $\mathcal{T}'$.

Attribute classifier. As training data for this classification task, we take the set of attribute names from the available real-world event logs used in our evaluation (see Section 3.4.1 for further details), complemented with attribute names from the *schema.org* vocabulary. This latter resource provides suggestions for standard attribute names that cover a broad range of object-related terms, e.g., *product*, as well as status and resource-related terms, e.g., *pending* or *agent*. Table 3.3 provides an overview of the training data obtained in this manner, which provides a good basis to train a classifier for our purpose.

Table 3.3: Characteristics of the data used to training our attribute classifier.

Source	$\texttt{object}$	$\texttt{object}_{status}$	$\texttt{action}_{status}$	$\texttt{actor}_{role}$	$\texttt{other}$
Real-world logs	6	6	1	8	38
Schema.org	68	22	4	91	70
Total	74	28	5	99	108

Using this data, we built a multi-class text classifier function $\texttt{classify}(d)$, which, given an attribute d, returns $t_d \in \mathcal{T}' \cup \{\texttt{other}\}$ as the semantic component type closest to $\texttt{name}(d)$, with $\texttt{conf}(t_d) \in [0, 1]$ as the respective confidence value. To operationalize the `classify` function, we encode the training data using the *GloVe* [141] vector representation for words. Subsequently, we train a *logistic regression* classifier on the obtained vectors, which can then be used to classify unseen attribute names. Since *GloVe* provides a state-of-the-art representation to detect semantic similarity between words, the classifier can recognize that, e.g., an `item` attribute is more similar to `object` attributes like *product*, than to the names associated with other component types in the dataset.

Detecting status attributes. Although the `classify` function is able to recognize the majority of relevant attribute classes, we observe that it relatively often fails to recognize $\texttt{object}_{status}$ attributes, which may be falsely classified as either `object` or `other`. A primary reason for this is that examples of the $\texttt{object}_{status}$ class are

underrepresented in the available training data, whereas the class also relates to a broad range of different kinds of statuses. However, we also observe that such $\texttt{object}_{status}$ attributes commonly follow a particular style. Specifically, status attributes often have a name that contains the past participle of a verb, e.g., *selected*, *is closed*, *accepted*, accompanied by Boolean or categorical attribute values, e.g., indicating that the respective object is indeed closed.

Based on this insight, we re-assign the class of any attribute $d \in \mathcal{D}_L^M$ of which $\texttt{name}(d)$ ends with a past participle, thus overwriting the class t_d assigned by the classifier with $\texttt{object}_{status}$. We detect such cases using standard NLP tools, such as *spaCy* [97]. In this manner, our approach annotates the values of the Boolean `Accepted` attribute of e_1 as $\texttt{object}_{status}$.

Classifying noun-only attributes in $\mathcal{D}_L^N$. Next, we turn to the classification of the noun-only attributes in $\mathcal{D}_L^N$, which were identified in the instance-level labeling step. Recall that these are textual attributes of which all values were entirely classified as being of the `object` component type, a situation that hints at a lack of context for their proper analysis (see, e.g., the `User` attribute of event e_2). To properly classify such attributes, we therefore first apply the same classifier as used for miscellaneous attributes. If $\texttt{classify}(d)$ provides a classification with a high confidence value, i.e., $\texttt{conf}(t_d) \geq \tau$ for a threshold τ, our approach uses t_d as the component type for an attribute $d \in \mathcal{D}_L^N$. In this way, we directly recognize cases where $\texttt{name}(d)$ is equal or highly similar to the attributes in our training data. However, if the classifier does not yield a confident result, we instead analyze the textual values in $\texttt{dom}(d)$.

Since noun-only attributes were previously re-assigned due to the lack of context available for their values, e.g., they just consist of words like *user* or *vendor*, we overcome this issue by placing them in artificial contexts that cover various kinds of semantic components, allowing us to recognize the appropriate role of an attribute value. To provide these contexts, we use a selection of highly expressive textual attribute values from the training collection of the instance-level labeling step. Specifically, we use a set N of texts consisting of the 891 unique attribute values that contain at least three different kinds of semantic components.

To illustrate this, consider *vendor* as an attribute value, and n = *confirm to customer that paperwork is ok* as the context value from N, which contains information on action (*confirm*), a passive resource (*customer*), an object (*paperwork*), and the object's status (*ok*). As shown in Figure 3.3, we create artificial texts for the attribute value by replacing a semantic component type from n with the word *vendor*. We subsequently feed these artificial texts into the language model used for instance-level labeling (Section 3.3.3), which can then be used to quantify the confidence score that the attribute value corresponds to the semantic component

type it replaced in an artificial text. For instance, in Figure 3.3, *vendor* is regarded as most likely corresponding to a passive resource for the context n, which we consider as a vote in favor of the $\texttt{passive}_{role}$ component type. Having computed these confidence scores for all attributes values in $\texttt{dom}(d)$ against all exemplary contexts in N, we assign $t_d \in \mathcal{T}' \cup \{\texttt{other}\}$ as the type that received the most votes overall.

Context *n*:
confirm (`action`) to customer ($\texttt{passive}_{role}$) that request (`object`) is ok ($\texttt{object}_{status}$)

Replace `action`:		
vendor to customer that request is ok	$\rightarrow \texttt{conf}(\texttt{action})$	= 0.26
Replace $\texttt{passive}_{role}$:		
confirm to *vendor* that request is ok	$\rightarrow \texttt{conf}(\texttt{passive}_{role})$	= **0.81**
Replace `object`:		
confirm to customer that *vendor* is ok	$\rightarrow \texttt{conf}(\texttt{object})$	= 0.68
Replace $\texttt{object}_{status}$:		
confirm to customer that request is *vendor*	$\rightarrow \texttt{conf}(\texttt{object}_{status})$	= 0.66

Figure 3.3: Insertion of value *vendor* into an existing context n, providing support of its being a passive resource ($\texttt{passive}_{role}$).

Recognizing instance-level attributes. Because we only focused on the type-level components $\mathcal{T}'$ in the above, we lastly check for every attribute that was classified as being resource related, i.e., with $t_d \in \{\texttt{actor}_{role}, \texttt{passive}_{role}\}$, if it actually corresponds to an instance-level component type instead. Particularly, we change t_d to the corresponding instance-level component type if $\texttt{dom}(d)$ has values that contain a numeric part or only consist of named-entities (e.g., *Pete*). For instance, a `User` attribute (cf. event e_2) with values like *batch_00*, contains numeric parts and is, thus reassigned to $\texttt{actor}_{instance}$, while the attribute of event e_3, with $\texttt{dom}(\texttt{org:resource}) = \{\textit{staff member}, \textit{system}\}$, clearly does not describe resource instances and, therefore, will retain its $\texttt{actor}_{role}$ component type.[5]

Component Identification Output

Having completed both the instance-level labeling and attribute-level classification steps, the component identification stage of our approach returns a collection of tuples (t, v) with $t \in \mathcal{T}$ a semantic component type and v a value, for each event $e \in E_L$. For values of a textual attribute $d_1 \in \mathcal{D}_L^T \setminus \mathcal{D}_L^N)$, v corresponds to part of the attribute value $e.d_1$. For those attributes that were classified at the attribute level, i.e., $d_2 \in \mathcal{D}_L^M \cup \mathcal{D}_L^N$, a tuple receives the entire value. In case of Boolean

[5] This is an interesting case, as `org:resource` attributes are intended for capturing actor instances.

values, the attribute's name is used if the original value is *true*, if it is *false*, *not* is prepended to the attribute's name.

For event e_1 of our running example, we obtain the tuples (`action`, *create*) and (`object`, *offer*) based on instance-level labeling and ($\texttt{action}_{status}$, *notAccepted*) and ($\texttt{actor}_{instance}$, *User_28*) based on attribute-level classification. For e_2, we obtain (`action`, *transfer*) and ($\texttt{passive}_{role}$, *execution syst.*) based on instance-level labeling and ($\texttt{actor}_{instance}$, *batch_00*) based on attribute-level classification. Finally, for e_3, we obtain (`object`, *declaration*), ($\texttt{action}_{status}$, *final*), (`action`, *approve*), and ($\texttt{actor}_{role}$, *supervisor*) based on instance-level labeling and ($\texttt{actor}_{role}$, *staff member*) based on attribute-level classification.

3.3.4 Semantic Component Categorization

In this section, we describe the two steps of the component categorization stage: action categorization and resource categorization.

Action Categorization

In this step, our approach categorizes identified `action` components, stemming from the previous stage, according to the eight high-level action categories described in Section 3.2.2. In this manner, we are able to recognize which events in a log relate to similar kinds of process steps, such as events that create, modify, or combine objects. For instance, we recognize that the *transfer* action of event e_2 modifies the handled item, whereas the *approve* action of event e_3 refers to a decision about a declaration.

We tackle this categorization task by establishing a set of *reference actions*, derived from the same MIT Process Handbook [122] that defines the eight high-level action categories. Then, given an action identified in the log, we use these reference actions to determine the most suitable high-level category.

Table 3.4 provides an overview of the collection of reference actions RA established for this purpose, which corresponds to the actions found in the first four layers of the action hierarchy defined by the handbook. We note that lower parts of the hierarchy do not provide additional reference actions, but rather contain more specific versions. For example, given *retire* as as a reference action for *Destroy*, further layers of the hierarchy include *retire physical object* and *retire digital object* as specializations, whose inclusion would not help to categorize individual actions. It is important to remark that some reference actions are part of multiple high-level categories, given that an action can have different impacts depending on its context. For example, *document* is part of both the *Create* and *Preserve* categories,

as documenting something can both indicate the creation of a new informational object, as well as the preservation of information, such as writing down a decision.

Category	Exemplary reference actions	Count
Create	*build*, *duplicate*, *design*, *produce*, *document*	15
Destroy	*retire*, *dispose*, *eliminate*, *obliterate*, *depreciate*	7
Modify	*improve*, *update*, *complete*, *move*, *send*	24
Preserve	*wait*, *retain*, *store*, *document*, *package*	10
Combine	*group*, *organize*, *match*, *aggregate*, *link*	10
Separate	*disaggregate*, *divide*, *segment*, *diversify*, *extract*	10
Decide	*select*, *determine*, *assign*, *assess*, *approve*	20
Manage	*assign*, *organize*, *allocate*, *schedule*, *budget*	11

Table 3.4: High-level action categories and exemplary reference actions.

To categorize an identified `action` component ac, we first determine the most similar reference action $ra \in RA$. For this purpose, we again employ the *GloVe* vector representations for words [141]. Given these vectors, we compute the cosine similarity between all vector pairs (ac, ra'), with $ra' \in RA$, retrieving the reference action ra with the highest similarity to ac. We then categorize action ac according to the high-level category that ra is part of. For example, given $ac =$ *transfer*, we obtain $ra =$ *move* as the reference action, so that ac is accordingly recognized as belonging to the *Modify* category.

Note that if there are multiple reference actions with the same, highest similarity score for a given action ac (and those actions are part of different categories), or if the most similar reference action is part of multiple high-level categories, we break the tie by computing the similarity between ac and the high-level actions themselves, assigning ac to the category with the highest similarity score. For example, given the action *write*, we get *document* as the reference action, which is part of both the *Create* and *Preserve* categories. Because *write* has a higher similarity to the *create* than to *preserve*, we assign the action to the former category.

In this manner, we annotate the *create* action of e_1 with the *Create* action type, the *transfer* action of e_2 with *Modify*, and the *approve* action of e_3 with *Decide*.

Resource Categorization

Finally, our approach turns to the categorization of the resources identified in the events of a log, determining whether they correspond to `human` or `system` actors. For instance, we categorize the $\texttt{actor}_{instance}$ component *User_28* of event e_1 as `human` and the $\texttt{actor}_{instance}$ component *batch_00* of event e_2 as `system`. Depending on the event log or specific resource, the information that may be available to

perform this categorization can vary greatly, ranging from specific descriptions of actor roles, e.g., *supervisor* and *staff member* in event e_3, to only having information on identifiers, e.g., *batch_00* or or *User_28* in the other two example events. Therefore, we propose several strategies, exploiting different kinds of information.

Identifying individual actors. We perform this categorization task for each actor contained in a log, which means that we first determine the set of resources RS that performed the activities of events, including all information available on them. To achieve this, we populate RS with all distinct $\texttt{actor}_{instance}$ components, since these represent unique actors. We associate these with any $\texttt{actor}_{role}$ components found in an actor's events. For example, given an event with $\texttt{actor}_{instance}$ = *0011* and $\texttt{actor}_{role}$ = *supervisor*, we add the resource tuple rs_1 = ($\texttt{actor}_{role}$ = *supervisor*, $\texttt{actor}_{instance}$ = *0011*) to RS. For events without $\texttt{actor}_{instance}$ but with $\texttt{actor}_{role}$ information, we consider unique combinations of $\texttt{actor}_{role}$ components as additional actors and add them to RS. For instance, for event e_3, which has two $\texttt{actor}_{role}$ components but no instance information, we add rs_2 = ($\texttt{actor}_{role}$ = *supervisor*, $\texttt{actor}_{role}$ = *staff member*) to RS.

Categorization strategies. To categorize a resource, we apply up to four categorization strategies in a sequential manner: (1) WordNet-based categorization of $\texttt{actor}_{role}$ components, (2) named entity recognition of $\texttt{actor}_{instance}$ components, (3) BERT-based resource classification, and (4) execution time analysis. The underlying idea is that the earlier strategies are highly precise, but may not be applicable for every resource $rs \in RS$. Therefore, if a strategy yields a hit for rs, we use that outcome to categorize it, otherwise our approach moves to the next strategy.

WordNet-based categorization of $\texttt{actor}_{role}$ *components.* This strategy aims to categorize a resource by comparing its $\texttt{actor}_{role}$ component, if available, to categories established in the lexical database *WordNet* [131]. This database relates words to each other via semantic relationships, such as hypernymy. A *hypernym* is a more general term for a given word, e.g., *color* is the hypernym of *red*.

If the $\texttt{actor}_{role}$ value of a resource rs is included in WordNet, we use the hypernymy relation to check if the value has a hypernym that corresponds to *person* or *organization*, in which case we categorize rs as being `human`, or to *system*, *computer*, or *information*, in which case we categorize rs as a `system`. For example, this strategy detects that a *vendor* is a *person* and, thus, a `human` actor, whereas a *database* corresponds to *information*, and is thus categorized as a `system`.

While this strategy is highly precise, its applicability is impeded by the limited scope of WordNet, which only covers rather common English terms. As such, this strategy requires that a resource has a semantically meaningful $\texttt{actor}_{role}$ component and, furthermore, that this is not too domain-specific.

NER for identifying $\texttt{actor}_{instance}$ *components.* This strategy detects `human` resources by determining if information contained in the $\texttt{actor}_{instance}$ component

corresponds to a known name, e.g., *Pete* or *Verena*. For this purpose, we use a standard NER tool [97] to check if the available instance information is found to be a *person*. In this manner, we are able to recognize a broad range of given and surnames of involved human actors, though this strategy can naturally only be applied to event logs that have not been properly pseudonomymized.

BERT-based resource classification. If the two previous deterministic strategies do not yield a result, we next apply a probabilistic strategy that classifies a resource according to the textual information contained in its components (if any), as well as based on the activities that it performs. While the former speaks for itself, the idea of the latter is that the activities can indicate whether they are likely to be performed automatically, i.e., by a system, or not. For instance, a resource performing a *transmit data* activity is more likely to be a system than a resource performing a *conduct quality check* activity, which requires the expertise of a human.

We operationalize this step by fine-tuning BERT to the task of classifying textual fragments, consisting of actor descriptions and activity names, as either *system* or *human*. To train this model, we employ a dataset and gold standard established by Leopold et al. [111], which contains sets of activities that are classified as being performed in an automated or in a manual fashion. In this manner, the fine-tuned model will learn to distinguish texts that relate to automated activities performed by `system` actors and manual activities performed by `human` actors. Given a resource rs, we consider each unique activity label l of an event that rs performs and apply the fine-tuned BERT model on the string $l + \textit{by} + rs.\texttt{actor}_{role}$ (or just l if rs has no $\texttt{actor}_{role}$). For instance, with $l =$ *in transfer to execution sys* and $rs =$ batch, we feed *in transfer to execution sys by batch* to the classifier. The classifier recognizes this activity as being performed by a system, resulting in a vote for the `system` category. Afterwards, we assign the category of rs according to the most commonly predicted class across all unique labels that rs has executed in the log.

Naturally, this strategy requires activity labels with sufficient semantic information, i.e., activities that refer to an action and/or object. Therefore, we only apply this strategy for activity labels that contain at least one of these components. Alternatively, in case $\texttt{actor}_{role}$ components are available for a resource, we can use these as standalone input to the classifier if no activity labels with sufficient semantic information are available. If neither informative labels nor actor descriptions are available, we turn to the last strategy.

Execution time analysis. Finally, if none of the previous strategies can be applied, we use a final categorization heuristic. Specifically, we consider the execution times of the events that are performed by a given resource r. If these hint at instantaneous execution of process steps, i.e., the timestamps of an event and its predecessor are equal[6], we categorize r as a `system`, otherwise as a `human`.

[6]While the timestamp is at least specific to the second.

In our example, we categorize the actor of e_1, *User_28*, as human using BERT-based resource classification, which is based on the word *User* itself, in combination with the labels of the events that the actor performed. The actor of event e_2 is categorized as system, based on the same strategy, whereas the actor of e_3 is categorized as human based on WordNet, which relates both its $\texttt{actor}_{role}$ components, *supervisor* and *staff member*, via the hypernymy relation to the word *person*.

3.3.5 Output

The component identification stage of our approach returns a collection of tuples (t, v) with $t \in \mathcal{T}$ a semantic component type and v a value, for each event $e \in E_L$, as described in Section 3.3.3. Afterwards, the component categorization stage adds a tuple $(\texttt{action:type}, cat_a)$, with cat_a as a high-level action, for each identified action component, and a tuple $(\texttt{actor:type}, cat_r)$, with cat_r as either human or system, if applicable. To enable the subsequent application of process mining techniques, the implementation of our approach returns an augmented XES event log that captures these tuples as dedicated, additional event attributes. We, thus, do not override any attributes from the original log.

Note that we support different ways to handle cases where an event has multiple tuples with the same semantic component type, such as the *draft* and *send* actions stemming from a *draft and send request* label or *staff member* and *supervisor*, both $\texttt{actor}_{role}$ components stemming from the same event e_3. Particularly, users can choose to collect the values into one attribute, i.e., action = [*draft, send*] and $\texttt{actor}_{role}$ = [*staff member, supervisor*], or into multiple, uniquely-labeled attributes, i.e., action:0 = *draft*, action:1 = *send* and actor:role:0 = *supervisor*, actor:role:1 = *staff member*. Information on the respective action types is then analogously captured in one or more event attributes. Lastly, if multiple $\texttt{object}_{status}$ (or $\texttt{action}_{status}$) attributes exist that each have Boolean values, e.g., isCancelled and isClosed for the Hospital log [123], these are consolidated into a single attribute, for which events are assigned a value based on their original Boolean attributes, e.g., $\{\perp, \textit{isCancelled}, \textit{isClosed}\}$.

For the running example, we obtain the annotated events that are shown in Figure 3.4 as the final output of our approach, which make the semantic components available for process analysis techniques.

concept:name	O_Create Offer
time:timestamp	11-07-16 18:54
org:resource	User_28
Accepted	False
action:name	create
action:type	create
object:name	offer
object:status	notAccepted
actor:instance	User_28
actor:type	human

(a) Output for event e_1.

concept:name	SRM: In transfer to Execution Syst.
time:timestamp	02-01-18 14:53
User	batch_00
action:name	transfer
action:type	modify
passive:role	Execution Syst.
actor:instance	batch_00
actor:type	system

(b) Output for event e_2.

concept:name	Declaration final_approved by supervisor
time:timestamp	26-02-18 05:15
org:resource	staff member
action:name	approve
action:status	final
action:type	decide
object:name	declaration
actor:role:0	supervisor
actor:role:1	staff member
actor:type	human

(c) Output for event e_3.

Figure 3.4: Final output of our approach for the running example's events.

3.4 Evaluation

In this section, we describe the evaluation experiments we conducted to demonstrate the accuracy of our proposed event log annotation approach with respect to its ability to both identify and categorize semantic components. Section 3.4.1 presents the collection of 14 real-world event logs we use as a basis for these experiments, Section 3.4.2 describes the employed implementation and experimental setup, whereas the results are presented and discussed in Section 3.4.3. Finally, Section 3.4.4 highlights the usefulness of our approach though three application cases that each show a semantics-aware analysis option that our approach enables. To support reproducibility, the employed implementation, gold standard, and links to the event logs are all available through a repository.[7]

[7] `https://github.com/a-rebmann/semantic-annotation`

3.4.1 Data Collection

For our evaluation, we selected all real-world event logs publicly available in the common 4TU repository[8], excluding those capturing data on software interactions or sensor readings, given their lack of natural language content. For collections that included multiple event logs with highly similar attributes, i.e., BPI13, BPI14, BPI15 and BPI20, we only selected one log per collection, to maintain objectivity of the obtained results. Table 3.5 depicts the details on the resulting collection of 14 event logs. They cover processes of different domains, e.g., financial services, public administration, and healthcare. Moreover, they vary substantially in their number of activities, textual attributes, and miscellaneous attributes.

Table 3.5: Event log characteristics, with A_L as the set of activities in the respective log, $\mathcal{D}_L$ of data attributes, and $\mathcal{D}_L^T$ of textual data attributes.

Log name	**Ref.**	$\lvert A_L \rvert$	$\lvert \mathcal{D}_L \rvert$	$\lvert \mathcal{D}_L^T \rvert$	**Log name**	**Ref.**	$\lvert A_L \rvert$	$\lvert \mathcal{D}_L \rvert$	$\lvert \mathcal{D}_L^T \rvert$
BPI12	[64]	24	4	2	BPI20	[70]	51	5	4
BPI13	[169]	4	11	4	CCC19	[132]	29	9	4
BPI14	[65]	39	7	2	Credit Req.	[62]	8	4	3
BPI15	[66]	289	13	3	Hospital	[123]	18	20	2
BPI17	[67]	26	15	4	RTFM	[53]	11	15	2
BPI18	[68]	41	12	5	Sepsis	[124]	16	31	1
BPI19	[69]	42	4	2	WABO	[38]	27	6	2

3.4.2 Evaluation Setup

We conducted our evaluation experiments based on the following setup.

Implementation. We implemented our approach in the form of a Python prototype, which is publicly available through the aforementioned project repository and also includes a command that allows users to directly incorporate our approach in their Python projects through *pip* installation.

Our implementation uses the *PM4Py* [32] library to handle event logs, Pandas[9] for the data type preprocessing stage, the BERT base uncased pre-trained language model[10] as a foundation for the instance-level labeling step, and *GloVe* vector representations [141] to determine semantic similarity between words.

Gold standard. We established a gold standard in which we manually annotated the contents of all 14 event logs used in the evaluation. For the component iden-

[8] `https://data.4tu.nl/search?q=:keyword:%20%22real%20life%20event%20logs%22`

[9] `https://pandas.pydata.org`

[10] `https://github.com/google-research/bert`

tification stage, we annotated all unique textual values, for instance-level labeling, and attributes, for attribute-level classification with their proper semantic component types. For the component categorization stage, we labeled identified resource components as `system` if the description of the data set clearly stated which resources are systems or if explicit information about the resource type was available from the event log itself. The `action` components that our approach identified in the 14 evaluation logs were labeled with their action type according to the MIT Process Handbook [122].

Cross-validation procedure. The semantic component identification stage of our approach uses a language model in the instance-level labeling step and a classifier in the attribute-level classification step that are both, among others, trained on data from the same real-world event logs used in the evaluation. Therefore, to avoid biasing the results, we perform our evaluation experiments using *leave-one-out* cross-validation, in which we repeatedly train our approach using data from 13 event logs and evaluate it on the 14th. This procedure is repeated such that each log in the collection is considered as the test log once. For the component categorization, this procedure is not required, since the training data we use does not stem from the collection of evaluation logs.

Evaluation metrics. To assess the performance of our approach, we compare the annotations obtained using our approach against the established gold standard. Specifically, we report on the standard *precision*, *recall*, and the F_1-*score*. Note that for instance-level labeling, we evaluate correctness per chunk, e.g., if a chunk (*purchase order*, `object`) is included in the gold standard, both *purchase* and *order* need to be associated with the `object` component type in the result, otherwise, neither is considered correct.

Baselines. To place the results obtained by our approach into context, we compare them to those obtained by relevant baselines. While there is no baseline against which we can compare our entire work, we compare the accuracy of our instance-leveling step against an existing activity label parser [110] and we compare the improved version of our attribute-level classification step against its initial version [149]. The details of these comparison are described in the respective parts of the results discussion below.

3.4.3 Evaluation Results

In this section, we report on the accuracy of our approach when it comes to semantic component identification and categorization.

Component Identification Results

We assess the results of the component identification stage by first considering the individual instance-level labeling and attribute-level classification steps, followed by a discussion of the overall results.

Instance-level labeling results. Table 3.6 shows the results obtained when labeling the 625 unique textual attribute values included in the event logs, where the *Count* column reflects the respective number of times a component type occurred in the gold standard. The table shows that our instance-level labeling technique is able to identify semantic components in textual attributes with high accuracy, achieving an overall F_1-score of 0.91. The comparable precision and recall scores, e.g. 0.94 and 0.95 for `action` or 0.89 and 0.88 for `object`, each suggest that the approach can accurately identify components while avoiding false positives. This is particularly relevant, given that nearly half of the textual attribute values also contain information beyond the scope of the semantic component types considered here (as shown in Table 3.2, there are 291 textual parts marked as `other`). Due to this ability to recognize which parts of texts are actually relevant for the component identification task, our approach even performs well on complex values. For example, for the *t13 adjust document x request unlicensed* from the WABO log [38], our approach correctly recognizes the objects (*document*, *request*), the action (*adjust*) and status (*unlicensed*), while omitting the superfluous content (*t13* and *x*).

Table 3.6: Instance-level labeling results for the 625 unique textual attribute values.

Component	**Count**	**Prec.**	**Rec.**	F_1
`object`	583	0.89	0.88	0.88
$\texttt{object}_{status}$	31	0.85	0.77	0.78
`action`	630	0.94	0.95	0.94
$\texttt{action}_{status}$	27	0.85	0.81	0.82
$\texttt{actor}_{role}$	69	0.93	0.84	0.88
$\texttt{passive}_{role}$	19	0.84	1.00	0.91
Overall	1,359	0.91	0.91	0.91

Challenges. We observe that the primary challenge for our approach relates to the differentiation between relatively similar component types, namely between the two kinds of statuses, $\texttt{object}_{status}$ and $\texttt{action}_{status}$, as well as the two kinds of resources, $\texttt{actor}_{role}$ and $\texttt{passive}_{role}$. Making this distinction is particularly difficult in cases that lack sufficient contextual information or proper grammar. For example, an attribute value *denied* can refer to either type of status, whereas it is even hard for a human to determine whether the *create suspension competent authority* label describes *competent authority* as a primary actor or a passive resource.

Baseline comparison. To put the performance of the instance-level labeling step into context, we compared it to a state-of-the-art parser for process model activity labels, proposed by Leopold et al. [110]. For a fair comparison, we retrained our approach on the same training data as used to train the baseline (corresponding to the collection of process models in Table 3.2) and only assess the performance with respect to the recognition of *objects* and *actions*, since the baseline only targets these. Table 3.7 presents the results obtained in this manner for the activity labels from all 14 considered event logs.

Table 3.7: Comparison of our instance-level-labeling technique against a state-of-the-art label parser. Both techniques are trained on process model activity labels and evaluated on the activity labels in our data collection.

		Our approach			**Baseline** [110]		
Component	**Count**	**Prec.**	**Rec.**	F_1	**Prec.**	**Rec.**	F_1
`object`	562	0.65	0.68	0.66	0.40	0.40	0.40
`action`	618	0.86	0.81	0.83	0.59	0.48	0.53
Overall	1,180	0.76	0.75	0.75	0.50	0.44	0.47

The table shows that our approach outperforms the baseline by a large margin, achieving an overall F_1-score of 0.75 versus the baseline's 0.47. Post-hoc analysis reveals that this improved performance primarily stems from activity labels that are more complex (e.g., multiple actions, various semantic components or compound nouns spanning multiple words) or lack proper grammar. This is in line with expectations, given that the baseline approach has been developed to recognize several established labeling styles, whereas we observe that event data often does not follow such modeling guidelines in practice. Finally, it is worth observing that the performance of our approach in this scenario is considerably lower than when trained on the full data collection (e.g., an F_1 of 0.66 versus 0.88 for the `object` component type), which highlights the benefits of our data augmentation strategies as well as the benefits of also training on activity labels besides process model activities.

Attribute-level classification results. After discarding 61 out of the total of 156 attributes in the preprocessing step and handling 24 attributes at the instance-level, a total of 71 attributes reach the attribute-level classification step. 36 of these attributes relate to one of the semantic component types, whereas the remaining 35 are of the `other` category. As shown in Table 3.8, our approach achieves highly accurate results for this step, with an overall precision of F_1 score of 0.92 for the 36 attributes corresponding to semantic components and of 0.91 for the entire set. Notably, these results reveal that our approach avoids false positives well, even though

a substantial amount of attributes are beyond the scope of our semantic component types, such as monetary amounts or timestamps. This can largely be attributed to the domain analysis employed in our approach's first step.

Table 3.8: Results of the attribute-classification step for the non-textual attributes.

		Our approach			Old approach [149]		
Component	**Count**	**Prec.**	**Rec.**	F_1	**Prec.**	**Rec.**	F_1
`object`	6	0.67	1.00	0.80	1.00	0.33	0.50
$\texttt{object}_{status}$	6	0.83	0.83	0.83	0.50	0.33	0.40
$\texttt{action}_{status}$	6	1.00	1.00	1.00	1.00	1.00	1.00
$\texttt{actor}_{instance}$	18	0.95	1.00	0.97	0.95	1.00	0.97
`other`	35	0.94	0.86	0.90	0.81	0.92	0.86
Overall (without `other`)	36	0.89	0.97	0.92	0.89	0.78	0.80
Overall (with `other`)	71	0.92	0.91	0.91	0.85	0.88	0.83

Note that the outstanding performance of our approach with respect to the $\texttt{action}_{status}$ and $\texttt{actor}_{instance}$ component types is in part due to the usage of standardized XES names for some of these attributes, enabling easy recognition. Yet, this is not always the case: 5 out of 18 $\texttt{actor}_{instance}$ attributes use different names than the XES standard (`org:resource` or `org:group`), such as `User` or `Assignment_Group`. Nevertheless, our approach maintains a high accuracy for these cases, correctly recognizing all such attributes. Overall, however, it is important to consider that these results were obtained for a relatively small set of 36 attributes. Thus, both the remarkable performance for most component types and the comparably low accuracy for `object` attributes must be considered with care.

Baseline comparison. Compared to our original attribute-level classification technique [149], we improved this step through the incorporation of additional training data and by adding an additional heuristic technique to improve the detection of $\texttt{object}_{status}$ attributes (see Section 3.3.3). As shown in Table 3.8, these adaptations resulted in an improved classification accuracy, raising the F_1 score from 0.80 to 0.92 for the semantic attributes. The increase in F_1 for the $\texttt{object}_{status}$ attributes from 0.40 to 0.83 highlights the value of the heuristic technique, whereas also our approach's ability to detect `object` attributes improved, with an F_1 of 0.80 versus 0.50 before.

Overall component identification results. The overall performance of the component identification stage can be considered as the average over the instance-level labeling and attribute-level classification results, weighted against the number of entities that were annotated with this component, i.e., a unique textual attribute value (instance-level) or an entire attribute (attribute-level). These scores are shown

in Table 3.9, which are naturally skewed towards the performance of the instance-level labeling step, given that this step covered 1,359 out of the 1,395 entities.

Table 3.9: Overall results of the component-identification stage.

Component	**Count**	**Prec.**	**Rec.**	***F*$_1$**
`object`	589	0.89	0.88	0.88
`object`$_{status}$	37	0.85	0.78	0.79
`action`	630	0.94	0.95	0.94
`action`$_{status}$	33	0.88	0.84	0.85
`actor`$_{role}$	69	0.93	0.84	0.88
`actor`$_{instance}$	18	0.95	1.00	0.97
`passive`$_{role}$	19	0.84	1.00	0.91
Overall	1,395	0.91	0.91	0.91

We observe that the approach achieves highly accurate overall results, with a precision, recall, and F_1-score of 0.91. Still, when considering the results per component type, we observe that there are considerable differences. These are largely due to the lower accuracy achieved for the underrepresented component types in the data set, as it is clear that our approach is highly accurate on more common component types, such as the F_1 score of 0.94 for recognizing actions.

Component Categorization Results

This section provides the results of the component categorization experiments. First, the results of the action categorization are discussed, before focusing on the categorization of resources.

Action categorization. Table 3.10 shows the results of the categorization of the 235 `action` components identified by our approach in the 14 evaluation logs per action type, consisting of 42 *create*, 4 *destroy*, 113 *modify*, 14 *preserve*, 6 *combine*, 5 *separate*, 49 *decide*, and 2 *manage* actions.

Overall, our approach rather accurately categorizes the identified actions, achieving an F_1 score of 0.79. For more common action types, our approach performs well, achieving an F_1 score of > 0.73 for *create*, *modify*, and *decide*. However, for the less common action types, i.e., *combine*, *destroy*, *manage*, *preserve*, and *separate* results vary, with an F_1 from 0.57 (*separate*) to 1.00 (*manage*).

An in-depth look into specific cases reveals that our approach is able to categorize both rather common actions, e.g., such as *generate*, *accept*, and *notify*, as well as actions performed in specialized processes, such as surgery [132]. For instance, the action *anesthetize* is correctly categorized as *modify*, whereas the action widen

is correctly categorized as *separate*. Though, this does not hold in all cases. For instance, the action *clean* is categorized as *create* rather than *modify*, which can be attributed to the limited amount of available training data.

Table 3.10: Results of the action-categorization step.

Category	**Count**	**Prec.**	**Rec.**	F_1
Create	42	0.70	0.90	0.79
Destroy	4	0.75	0.75	0.75
Modify	113	0.85	0.84	0.85
Preserve	14	0.62	0.62	0.62
Combine	6	0.62	0.83	0.71
Separate	5	0.44	0.80	0.57
Decide	49	0.91	0.61	0.73
Manage	2	1.00	1.00	1.00
Overall	235	0.81	0.79	0.79

A challenge of this task is the ambiguity of some of the actions. For instance, *suspension* could be considered as either a *destroy*, *modify*, or a *decide* action, making it difficult to categorize such actions. A related challenge is that the framework does not classify actions into disjoint top-level categories. For instance, the action *document* is both categorized as *create* and *preserve*, which both makes sense given that an artifact can be considered both as preserved or created when it is documented. Similarly, the action *allocate* is both categorized as *manage* and *decide*. While such framework-specific issues are problematic, even with this set-up, our approach provides a helpful categorization of actions and, thus, activities.[11]

Resource categorization. The 14 evaluation logs contain a total of 5,236 distinct resources. 5,204 of these are `human`, while only 32 are `system`, an imbalance that clearly reflects the fact that many different human actors can be involved in a process, while the number of different systems is typically limited.

Overall, we achieve an F_1 score of 0.999 for `human` (precision and recall also 0.999) and of 0.80 for `system` actors (with a precision of 0.86 and recall of 0.75). The performance of the individual strategies in terms of their number of *hits*, i.e., how often they were applicable, and their precision is depicted in Table 3.11.

As shown in the table, the WordNet-based strategy has perfect precision, but is only applied to a few cases. However, it should be noted that these eight hits correspond to entire resource classes (i.e., $\texttt{actor}_{role}$ components), rather than individual resources like the other strategies. For seven of these cases there are no instances

[11] We demonstrate this usefulness in an application case in Chapter 4.

Table 3.11: Performance of the resource-categorization strategies, with the asterisks (*) indicating class-level hits.

Strategy	Category	Hits	Prec.
WordNet	human	6*	1.00
	system	2*	1.00
NER	human	585	1.00
	system	0	-
BERT	human	1,340	0.99
	system	30	0.85
Time	human	1,299	1.00
	system	0	-

contained in the log, such as the *employee*, *director*, and *supervisor* roles in the BPI20 log. However, the *vendor* role from the BPI19 log relates to 1,975 different resources, thus highlighting the overall relevance of this class-level strategy.

For the NER-based strategy, we again observe a perfect precision, though this strategy can only be applied to the BPI13 log. This event log uses first names, such as *Tomas*, *Carrie*, and *Niklas*, to refer to 585 specific resources.

The BERT-based strategy, which primarily focuses on activity label names, can be applied more broadly than the previous strategies. Yet, as shown by the precision of 0.99 for `human` resources and 0.85 for `system` ones, the accuracy of this strategy is still high. An in-depth analysis of these results reveals that a primary challenge here involves activities that can be executed by both human and system resources, such as seen for the BPI18 [68] log. In these situations, the analysis of activity labels does not always allow for the appropriate distinction between resource types, affecting the strategy's precision.

Finally, the results for the heuristic analysis of execution times are rather inconclusive, since this strategy was not applicable to `system` resources in the employed event log collection. However, the strategy also did not falsely categorize a `human` resource as a system, thus nevertheless achieving a precision of 1.00.

3.4.4 Application Cases

To highlight the benefits of our approach, we next look at a selection of three application cases that our approach enables, involving real-world event logs. Specifically, we show how the semantic information identified by our approach can support (1) activity refinement, (2) object-centric process analysis, and (3) the analysis of a process' automation degree.

Activity refinement. In this first application case, we show how semantic components identified in the instance-level labeling step of our approach can be used to establish more appropriate activities for the *permit log* from the BPI Challenge 2020 [70]. This log consists of 7,065 cases and 86,581 events, divided over 51 activities (according to the activity label, i.e., the `concept:name` attribute). This relatively large number of activities is problematic when aiming to gain insights about the recorded process. Particularly, any process model derived on its basis will automatically exceed the recommended maximum of 50 nodes in a process model [129] and quickly reach a spaghetti-like structure.

However, an assessment of the activity labels and the semantic components identified in them, reveals that the labels in the log are polluted by superfluous information, resulting in an unnecessarily high number of different activities. Specifically, the majority of activity labels mixes up information about the conducted activity, which should indeed be contained in a label, with information on the actor that performs the activity, which should rather be captured in a dedicated $\texttt{actor}_{role}$ attribute. Typical examples of this situation are labels such as *declaration approved by budget owner* and *declaration approved by administration*.

Recognizing this situation, we can use the semantic components identified by our approach to establish refined activity labels, which consist of only the information from the `action` and `object` roles, e.g., *declaration approved*, while deferring the actor information to a $\texttt{actor}_{role}$ attribute. This operation yields an event log in which the number of activity labels has been greatly reduced, from 51 to just 21. In this way, we have consolidated different pieces of semantic process information in dedicated places, i.e., activity information in the label and actor information in a separate attribute, whereas, when used as the new activity, the refined activity labels allow for the discovery of smaller and hence more understandable process models. Finally, it is important to point out that this transformation does not lead to any loss of information, given that the old labels are not overwritten, whereas it is, as always, also possible to define activities as combinations of the (refined) activity labels plus the $\texttt{actor}_{role}$ attribute.

Object-centric process analysis. In this application case, we demonstrate how the semantic information identified by our approach can be used to obtain an object-centric view on a process, which helps to provide clearer insights into processes that deal with various kinds of objects. For this, we again use the *permit log* [70].

After applying our approach, we observe that the log contains six different objects (indicated as `object`): *permit*, *trip*, *request for payment*, *payment*, *reminder*, and *declaration*. Such information was not initially available in the log, given that these objects were identified in the activity labels themselves (see also the previous application case). Yet, after having identified them, we can investigate the execu-

tion of the process in both an inter-object and intra-object manner, which provides novel insights that the original log could not reveal.

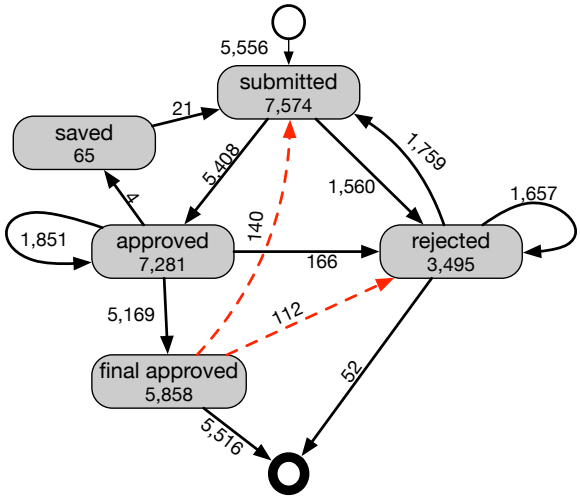


Figure 3.5: Example for object-centric analysis. The DFG shows the actions applied to the object *declaration* in the log (includes 100% activities, 50% paths).

To illustrate this potential, consider the DFG shown in Figure 3.5, which we obtained by selecting all events related to declarations, i.e., with `object` = *declaration*, and using the identified actions to establish the activity. The figure clearly reveals how these objects are handled in the process. Mostly, declarations are *submitted*, *approved*, and then *final approved*. Interestingly, though, we also see 112 cases in which a declaration was definitely approved, yet rejected afterwards. Furthermore, we see 140 cases, in which a declaration was resubmitted after it was already definitely approved.

It is important to stress that such insights would not be possible from the original log, given that, in reality, the events related to declarations may be interspersed with events related to other objects. Furthermore, by employing different filters and activities, this object-centric view can be used as a basis for a wide range of other insights, e.g., to reveal how documents are handed over between different employees or to reveal the inter-relations between objects. Besides using the annotated object-centic information directly, it can also serve as a starting point to transform existing classical event logs into object-centric ones [156], which can in turn be used for object-centric process mining [6].

Analysis of automation degree. Finally, we use the resource categorizations provided by our approach to assess the automation degree and impact of system resources for the *purchase order* event log, part of the BPI Challenge 2019 [69]. The event log contains 251,734 cases, 1,595,923 events, and 628 unique resources.

By applying the resource categorization step of our approach, we discover that the events in the log originate from 608 `human` resources and 20 `system` resources, primarily captured in the log's `User` attribute. We can leverage this categorization, stored in a `resource:type` event attribute, to analyze the automation degree of the Purchase Order process. At the event level, we find that about 79% of the events in the log are performed by employees, whereas the other 21% are performed in an automated manner, i.e., by a `system`. When considering full traces, we recognize that just 2,338 cases (0.9%) are entirely performed by systems, showing that only a fraction of cases can be handled in an automated manner. By contrast, we find that 162,505 (64.9%) do not involve any automatically executed steps. Both insights, thus, hint at clear opportunities for further automation, e.g., through the application of robotic process automation [109] technology.

3.5 Limitations

The evaluation results demonstrate that our annotation approach can be used to obtain semantic information about events that are otherwise not readily available and thus cannot be taken into account by process mining techniques. Nevertheless, it is important to consider these outcomes in light of certain limitations. In particular, there are limitations related to our approach and the evaluation.

The main limitation of our approach is its dependency on the availability of natural language information in event logs. All real-world event logs we have at our disposal show at least some natural language as part of the *activity* attribute and attribute names. However, it is imaginable that some organizations rather use highly domain-specific terminology, abbreviations, and codes in their event data. In such cases, our approach is not applicable in a straightforward manner. Dictionaries for such specific information could be used for a translation into natural language labels and attribute names, though. Another limitation relates to our action-categorization step, which relies on static embeddings of the actions. As a consequence, it cannot differentiate between different contexts in which an action is applied, which may lead to incorrect categorization results. In particular, incorrect categorizations may occur if the action does not fully capture what is being done by itself. For instance, consider the activity *make selection*. Its action, *make*, is incorrectly categorized as *create* because what is done is in fact captured by the object, *selection*, and the action should, thus, rather be categorized as *select*. To avoid this, contextualized embeddings of the entire activity label could be used instead. This would lead to less transparency of the results, though, because we would omit the action taxonomy that we use as a basis now.

As for our evaluation, even though we used a broad range of real-world event logs that cover diverse process domains and have various labeling styles, we cannot conclude that our approach works equally well on event data from unseen domains. Furthermore, there is limited evaluation data available for assessing our attribute classification step, whereas the data for resource categorization is highly unbalanced, comprising a considerable amount of samples for the `human` class and little for the `system` class. Here, further evaluation data could help to provide stronger performance propositions.

3.6 Related Work

Our work relates to research streams focused on the analysis of activity labels, semantic annotation of process information, and semantic role labeling.

3.6.1 Activity Label Analysis

Various approaches strive to either disambiguate or consolidate labels in event logs. Lu et al. [119] propose an approach to detect duplicate activity labels, i.e., labels that are associated with events that occur in different contexts. By refining such duplicates, the quality of subsequently applied process discovery algorithms can be improved. By contrast, Sadeghianasl et al. [163, 164] aim to detect the opposite case, i.e., situations in which different labels are used to refer to behaviorally equivalent events. They achieve this through context-aware metrics [164] and crowdsource-based gamification [163]. As already discussed in Section 2.2.4, other approaches strive for the analysis of labels, such as work by Deokar and Tao [55], which group together activities that are semantically similar, as well as the label parsing approach by Leopold et al. [110] against which we compared our work in the evaluation. Other research aims to improve event log quality by relabeling activities based on proposed quality metrics. To this end, they suggest NLP tasks, such as POS-tagging, based on the quality they computed for a log [147].

3.6.2 Semantic Annotation of Process Information

Various works are complementary to ours, as they also strive to annotate event data or process models with different kinds of semantic information. Work by Tsoury et al. [176] strives to augment logs with additional information derived from database records and transaction logs. Works by Leopold et al. focus on the categorization of process activities, achieved by mapping process model components to an existing process categorization [114] and by categorizing activities according to their degree of automation [111].

3.6.3 Semantic Role Labeling

Beyond the scope of process analysis, our work relates to semantic annotation applied in various other contexts. Most prominently, SRL is a widely recognized task in NLP (see also Section 2.2.3), which labels spans of words in sentences that correspond to semantic roles. While early work in this area mostly applied feature engineering methods [146], recently deep learning-based techniques have been successfully applied, e.g., [95, 188]. In the context of web mining, semantic annotation focuses on assigning semantic concepts to columns of web tables [189], while in the medical domain it is, e.g., used to extract the symptoms and their status from clinical conversations [71].

3.7 Summary

In this chapter, we proposed an approach for the automatic semantic annotation of event data. Namely, our approach identifies up to eight semantic component types per event, covering objects, actions, actors, and other resources, without imposing any assumptions on the structure of an event log's attributes. It then further categorizes the identified action and resource components into predefined categories, enabling new analysis opportunities that consider the meaning of events.

We demonstrated our approach's efficacy through evaluation experiments using a wide range of real-world event logs. The results show that our approach accurately identifies the targeted semantic components from textual attributes, whereas our attribute classification techniques yield good results when dealing with the information contained in non-textual attributes. In both cases, we showed that our techniques outperform existing state-of-the-art work. Furthermore, our approach performs well in further categorizing identified semantic components. Finally, we highlighted the potential of our work by illustrating some of its benefits in three application cases based on real-world data. Particularly, we showed how our approach can be used to refine and consolidate activities in the presence of polluted labels, as well as to obtain object-centric insights about a process. Moreover, we showed that detailed analysis of the automation degree of a process is enabled by categorizing resource components.

Chapter 4

Constraint-Driven Abstraction of Event Logs

Events that are recorded during the execution of a process are often too fine-granular to be used for process mining. This is because fine-granular events lead to a considerable amount of different activities and a high degree of behavioral variability in the traces of a given event log, causing process mining results to become exceedingly complex and challenging to interpret [187]. Consequently, an organization cannot improve its understanding of the underlying process based on these results. To tackle this issue, event-abstraction approaches lift the traces of an event log to a more abstract representation, by grouping low-level events into higher-level ones. Existing approaches (cf. [60, 187]) differ in the adopted algorithms and employ, e.g., temporal clustering of events [51] or the detection of predefined patterns [125].

Although these approaches can help reduce the behavioral variability in an event log and thus, the complexity of process mining results, their focus is on *how* the abstraction is conducted, rather than *what* properties the resulting log should satisfy. However, without any control on the result of event abstraction, it is hard to ensure that this result is appropriate for a specific analysis goal. Therefore, to provide effective abstraction, an approach must enable users to incorporate dedicated constraints on the resulting event log.

In this chapter, we achieve this by proposing an event-abstraction approach that enables a user to impose requirements on the resulting log in terms of constraints. As such, it supports a declarative characterization of the properties that the abstracted log should adhere to, in order to ensure a specific analysis goal. In particular, we define the optimal-event-abstraction task that requires minimizing the distance to the original log while satisfying a set of constraints on the abstracted log.

To satisfy this task, our approach covers a broad set of common constraint types and a distance measure. As part of our approach, we present an algorithm for exhaustive event abstraction. Striving for more efficient processing, we also provide a heuristic algorithm that is guided by behavioral dependencies found in the log. Our evaluation shows that the event logs obtained with our approach provide better abstraction and are more cohesive than those obtained with baseline approaches. As such, process discovery algorithms also yield more structured models.

This chapter is based on a paper titled "GECCO*: Constraint-driven Abstraction of Low-level Event Logs*" [157] by Adrian Rebmann, Matthias Weidlich, and Han van der Aa.

In the remainder of the chapter, Section 4.1 first motivates the need for user-defined constraints in event abstraction. Next, Section 4.2 describes the scope of our approach by defining the optimal-event-abstraction task, specifying covered constraint types, and defining a distance function. Section 4.3 then presents our abstraction approach and Section 4.4 reports on evaluation results, which we complement with two application cases. Section 4.5 reflects on limitations of our work. Finally, Section 4.6 reviews related work, before Section 4.7 provides a summary.

4.1 Problem Illustration

For illustration purposes, consider the event log in Table 4.1, which consists of the activity sequences of four traces that correspond to a request-handling process. Blue underlined events denote activities performed by a clerk, whereas the others are performed by a manager.

Table 4.1: Exemplary traces of an event log.

Trace	Activity sequence
σ_1	⟨rcp, ckc, acc, prio, inf, arv⟩
σ_2	⟨rcp, ckt, rej, prio, arv, inf⟩
σ_3	⟨rcp, ckc, acc, inf, arv⟩
σ_4	⟨rcp, ckc, rej, rcp, ckt, acc, prio, arv,inf⟩

The event log shows that each case starts with the receipt of a request (*rcp*) by a clerk. The clerk checks the request either casually (*ckc*) or thoroughly (*ckt*) depending on the information provided. Then, the request is forwarded to a manager, who either accepts (*acc*) or rejects (*rej*) it. Afterwards, the clerk may or may not assign priority to a request (*prio*), before they inform the customer (*inf*) and archive the request (*arv*). The latter two activities can be performed in either order,

as shown, e.g., in σ_1 and σ_2. As shown in trace σ_4, a rejected request may also be returned to the applicant, who will resubmit it, restarting the procedure.

Although this process consists of only eight distinct activities, its behavior is already fairly complex. This is evidenced by the DFG shown in Figure 4.1. The graph's complexity already obscures key behavioral aspects of the process. Event abstraction may alleviate this problem. However, existing techniques focus on how the abstraction should be done. For instance, they may exploit that the steps *ckt*, *ckc*, *acc*, and *rej* are closely correlated from a behavioral perspective and abstract them to a single activity. Yet, this is not meaningful for many analysis tasks, as it would obscure the fact that the activity encompasses some steps performed by a clerk (*ckt* and *ckc*), whereas others are performed by a manager (*acc* and *rej*).

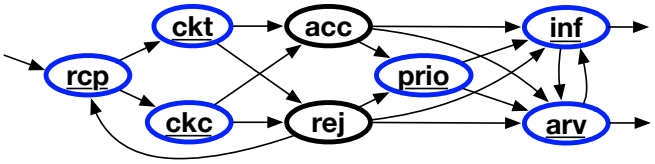


Figure 4.1: DFG of the running example.

By incorporating user-defined constraints on what properties the abstracted log should satisfy, such a result can be avoided. For instance, if a user wants to primarily understand the interactions between employees, while abstracting from details on the individual steps they perform, a constraint may enforce that each activity comprises only events performed by the same employee role. If applied in a naive manner, this constraint would result in two groups of activities, i.e., $g_{clrk} = \{rcp, ckc, ckt, prio, inf, arv\}$ and $g_{mgr} = \{acc, rej\}$. Yet, using these groups for abstraction is not meaningful either. The group g_{clrk} includes activities that occur at the start of the process, as well as ones that only happen at the end. Moreover, abstracting the steps in g_{mgr} to a single activity would obfuscate that $\{acc, rej\}$ exclude each other and that only after activities *rej*, the process may be restarted.

Against this background, our approach aims at constructing activities for groups of events that satisfy user-specified constraints, while also preserving the behavior represented in traces as much as possible. For the example, this would result in an abstraction that consists of four groups: $g_{clrk1} = \{rcp, ckc, ckt\}$, containing the initial activities performed by the clerk, $\{acc\}$ and $\{rej\}$, as singleton groups of activities that are mutually exclusive and both performed by the manager, and $g_{clrk2} = \{prio, inf, arv\}$, the final activities of the process performed by the clerk. The DFG obtained with this abstraction is shown in Figure 4.2. It highlights that a clerk starts working on each case, before handing it over to the manager. Accepted requests are completed by the clerk, whereas rejected requests may be completed or returned to the start of the process.

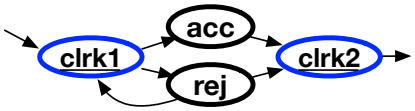


Figure 4.2: DFG of the log abstracted with our approach.

4.2 Scope

This section describes the scope of our approach by defining the optimal-event-abstraction task (Section 4.2.1), specifying covered constraint types (Section 4.2.2), and defining a distance function (Section 4.2.3).

4.2.1 Event-Abstraction Task

Event abstraction aims to group similar events for an event log. Formally, this task is captured by a grouping, i.e., a set of groups, $G \subset 2^{\mathcal{A}}$, over the activities of a given log L, A_L, such that each activity $a \in A_L$ is part of exactly one group $g \in G$. Given a grouping, a function $\texttt{abstract} : 2^{\mathcal{E}^*} \times 2^{\mathcal{A}} \rightarrow 2^{\mathcal{E}^*}$ is applied to obtain an abstracted log L'. For instance, using the log and grouping from Section 4.1, σ_1 is abstracted to $\sigma_1' = \langle \textit{clrk1}, \textit{acc}, \textit{clrk2} \rangle$.

We target scenarios in which a user formulates requirements on what properties the abstracted event log and, hence, the grouping G should satisfy, e.g., to group only events performed by a single role (see Section 4.1). Then, we aim to identify a grouping $\hat{G}$ that meets these requirements, while preserving the behavior of the traces as much as possible. To this end, we define $\texttt{dist} : 2^{\mathcal{E}^*} \times 2^{\mathcal{A}} \rightarrow \mathbb{R}$ as a *distance function* quantifying the distance of a grouping to an event log. Also, using $\mathcal{C}$ to denote the universe of possible constraints, we define a predicate $\texttt{holds} : 2^{\mathcal{A}} \times \mathcal{C} \times 2^{\mathcal{E}^*} \rightarrow \{true, false\}$ to denote if a grouping satisfies a set of constraints for a given log. Based thereon, we define optimal event abstraction as follows:

Definition 10 (Optimal event abstraction) *Given an event log L with activities A_L, a distance function* dist*, and a set of constraints* C*, optimal event abstraction aims to find an optimal grouping $\hat{G} = \{g_1, \ldots, g_k\}$, such that:*

- *$\hat{G}$ is an exact cover of A_L, i.e., $\bigcap_i^k g_i = \emptyset \wedge \bigcup_i^k g_i = A_L$;*
- *$\hat{G}$ adheres to the desired constraints* C*, i.e., $\texttt{holds}(\hat{G}, C, L) = true$;*
- *the distance $\texttt{dist}(\hat{G}, \mathrm{L})$ is minimal.*

4.2.2 Covered Constraint Types

Our approach is able to handle a broad range of constraints on a grouping G. As shown through the examples in Table 4.2, we consider *grouping* constraints,

class-based constraints, and *instance-based* constraints. The table also indicates a monotonicity property of the constraints, which is important when aiming to find an optimal grouping in an efficient manner.

Table 4.2: Exemplary constraints covered by our approach.

Category	**Examples**	**Monotonicity**
Grouping constraints	There should be at most 10 groups in the final grouping.	n/a
	There should be at least 5 groups in the final grouping.	n/a
Class-based constraints	There should be at least 5 activities per group.	monotonic
	At most 10 activities should be grouped together.	anti-monotonic
	The activities *rcp* and *acc* cannot be members of the same group.	anti-monotonic
	The activities *inf* and *arv* must be members of the same group.	non-monotonic
Instance-based constraints	At least 2 distinct document codes must be associated with a group instance.	monotonic
	The cost of a group instance must be at most 500$.	anti-monotonic
	The duration of a group instance must be at most 1 hour on average.	non-monotonic
	The time between consecutive events in a group instance must at most be 10 minutes.	anti-monotonic
	Each group instance may contain at most 1 event per activity.	anti-monotonic
	At least 95% of the group instances must have a cost below 500$.	anti-monotonic

Grouping constraints. This constraint category can be used to bound the size of a grouping G, i.e., the number of high-level activities that will appear in the abstracted log. An upper bound restricts the size and complexity of the obtained log, whereas a lower bound can limit the applied degree of abstraction.

Satisfaction. We use $C_G \subseteq C$ to refer to the subset of grouping constraints. Whether a constraint $c \in C_G$ holds can be directly checked against the grouping size, $|G|$. As such, for the `holds` predicate, we require $\forall c \in C_G : c(G) = true$.

Class-based constraints. The second category of constraints can be used to influence the characteristics of an individual group (higher-level activity type) $g \in G$ in terms of the class-level attributes and, in particular, the (low-level) activities that it can contain. Our approach supports any class-based constraint for which satisfaction can be checked by considering g in isolation, i.e., without having to compare g

to other groups in G. As shown in Table 4.2, this, for instance, includes constraints that each group should comprise at least (or at most) a certain number of activities, as well as *cannot-link* and *must-link* constraints, which may be used to specify that two activities must or must not be grouped together.

Satisfaction. We use $C_C \subseteq C$ for class-based constraints. The satisfaction of a constraint $c \in C_C$ is directly checked by evaluating the contents of each group $g \in G$. Hence, the `holds` predicate requires that $\forall g \in G, \forall c \in C_C : c(g) = true$.

Monotonicity. Class-based constraints that specify a minimum requirement on groups, e.g., a minimal group size, are monotonic: If the constraint holds for a group g, it also holds for any larger group g', with $g \subset g'$. In other words, adding activities to a group can never result in a (new) constraint violation. By contrast, constraints that express requirements that may not be exceeded, e.g., a maximal group size or *cannot-link* constraint, are anti-monotonic: If they hold for a group g, they also hold for any subset of that group $g' \subset g$. However, if a group g violates a constraint, a larger group g', with $g \subset g'$, also violates it.

Instance-based constraints. The third category comprises constraints that must hold for each *instance* of a group $g \in G$, i.e., a sequence of (not necessarily consecutive) events that occur in the same trace and of which the activities are part of g. In line with Definition 1 that defines event attributes, we use $g.d$ to refer to the set of values of attribute d for a group g when defining constraints of this type.

As indicated in Table 4.2, diverse constraints can be defined on the instance-level, relating to attribute values, associated roles, and duration, such as *the total cost of an instance is at most 500$* and *the average duration of group instances must be at most 1 hour*. As shown in the table's last row, also looser constraints may be expressed, such as ones that only need to hold for 95% of the respective group instances. In fact, as for class-based constraints, our approach supports all constraints of which satisfaction can be checked for an individual group g.

Satisfaction. We write $C_I \subseteq C$ for the instance-based constraints. Contrary to the other categories, these constraints must be explicitly checked against the event log L, specifically for each group $g \in G$ and each instance of g in the traces of L.

Formally, we first define a function $\texttt{inst} : \mathcal{E}^* \times 2^{\mathcal{A}} \rightarrow 2^{\mathcal{E}^*}$, which returns all instances of a group in a given trace. The operationalization of `inst` is straightforward for simple cases: An instance of group g is the projection of the activities of g over a trace σ. In σ_1, σ_2, and σ_3 of our running example, exactly one instance of each group occurs per trace and, e.g., $\texttt{inst}(\sigma_1, g_{clrk1}) = \{\langle rcp, ckc \rangle\}$. However, processes often include recurring behavior, such as trace σ_4 with its activity sequence ⟨rcp, ckc, rej, rcp, ckt, acc, prio, inf, arv⟩, in which a request is first rejected, sent back to the restart the process, and then accepted in the second round. Here, to detect multiple instances of a group, we instantiate the function `inst`

based on an existing technique [5] that recognizes when a trace contains recurring behavior and splits the (projected) sequence accordingly. For the above trace, this yields $\texttt{inst}(\sigma_4, g_{clrkI}) = \{\langle rcp, ckc\rangle, \langle rcp, ckt\rangle\}$. Note that `inst` can also be used to enforce cardinality constraints, e.g., if a user desires that each group instance should contain at least 2 events of a particular activity.

Given the function `inst`, a constraint $c \in C_I$ is satisfied if for each group $g \in G$, c holds for each instance $\xi \in \texttt{inst}(\sigma, g)$, for each $\sigma \in L$. Note that constraints are vacuously satisfied for traces that do not include an instance of a particular group, i.e., where $\texttt{inst}(\sigma, g) = \emptyset$. Therefore, for instance-based constraints, `holds` is checked for each $g \in G$ as $\forall\, c \in C_I, \forall\, \sigma_i \in L, \forall\, \xi \in \texttt{inst}(\sigma_i, g) : c(\xi) = true$. For looser constraints, e.g., ones that should for 95% of the group instances, predicate satisfaction is adapted accordingly.

Monotonicity. As for class-based ones, instance-based constraints are monotonic when specifying a minimum requirement to be met, e.g., each instance should take *at least* one hour, and anti-monotonic when specifying something that may not be exceeded, e.g., each instance may take *at most* one hour. However, constraints in C_I may also be based on aggregations that behave in a non-monotonic manner, such as constraints that consider the *average* or *variance* of attribute values per instance or *sums* including negative values. In these cases, adding and removing activities from a group can result in a violated constraint to now hold or vice versa.

4.2.3 Distance Measure

To determine which activities are suitable candidates to be grouped together, we employ a distance function $\texttt{dist}(G, L)$ that quantifies the relatedness of the activities per group. Although our work is largely independent of a specific distance function, we argue that event abstraction should group together activities such that (1) events within a group are *cohesive*, i.e., the events belonging to a single group instance occur close to each other, meaning there are few interspersed events from other instances; (2) events within a group are *correlated*, i.e., the events belonging to a single group typically occur together in the same trace and group instance; (3) larger groups are favored over *unary groups*, i.e., the grouping G actually results in an abstraction. To capture these three aspects, we propose the following distance function for an individual group g and a log L:

$$\texttt{dist}(g, L) = \sum_{\xi \in \texttt{inst}(L,g)} \frac{\frac{\texttt{interrupts}(\xi)}{|\xi|} + \frac{\texttt{missing}(\xi,g)}{|g|} + \frac{1}{|g|}}{|\texttt{inst}(L, g)|} \tag{4.1}$$

The first summand in the numerator of Equation 4.1 considers cohesion. Here, $\texttt{interrupts}(\xi)$ counts how many events from other instances are interspersed be-

tween the first and last events of a given group instance ξ. As such, this penalizes groups of events that are often *interrupted* by others, e.g., in a trace $\langle a, b, c, d, e \rangle$, grouping a and e together is unfavorable, since the instance $\langle a, e \rangle$ has three interspersed events. In Equation 4.1, the number of interruptions is considered relative to the length of ξ. The second summand in the numerator of Equation 4.1 quantifies the degree of completeness of ξ with respect to g, thus capturing the correlation between the events in g. Here, $\texttt{missing}(\xi, g)$ returns how many activities from g are missing from its instance ξ, which is then offset against the total number of activities $|g|$. Finally, since groups with a single activity have perfect cohesion and correlation by default, we include $\frac{1}{|g|}$ to ensure that larger groups with the same cohesion and correlation are favored, thus avoiding unary groups when possible.

To quantify the total distance of a grouping G, we sum the distances of groups in G, resulting in the following function that our approach aims to minimize:

$$\texttt{dist}(G, L) = \sum_{g \in G} \texttt{dist}(g, L) \tag{4.2}$$

4.3 Abstraction Approach

In this section, we describe how our approach finds an optimal event grouping based on an event log, given the distance function and constraints defined in the previous section, to abstract that log. Section 4.3.1 provides a high-level overview, while Section 4.3.2 to Section 4.3.4 outline the algorithmic details.

4.3.1 Approach Overview

As shown in Figure 4.3, our approach takes an event log L and a set of user-defined constraints C as input and then applies three main steps in order to obtain an abstracted event log L'.

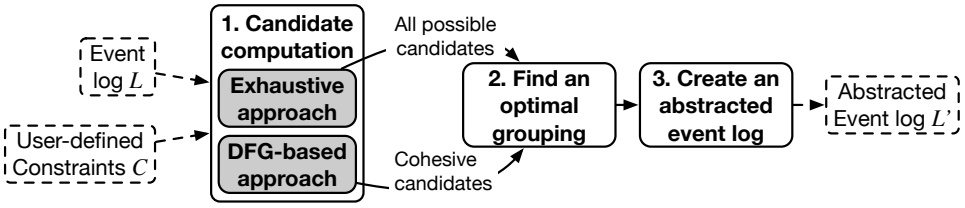


Figure 4.3: Overview of the abstraction approach.

In the first step, our approach computes a set of candidate groups $\mathcal{G}$, i.e., groups of activities that adhere to the constraints in C. As depicted in Figure 4.3, we propose two instantiations for this step: an exhaustive instantiation and an efficient

DFG-based one. The *exhaustive* instantiation yields a set of candidates that is guaranteed to be complete and, thus, assures that it can afterwards be used to establish an optimal grouping if it exists. Since this approach may be intractable in practice, we also propose a *DFG-based* alternative. It only retrieves cohesive candidates, i.e., candidates likely to be part of an optimal grouping. By exploiting the process-oriented nature of the input data, cohesive candidates are identified efficiently. Any solution obtained using this instantiation is still guaranteed to satisfy the constraints in *C*, yet may have a sub-optimal distance score.

In the second step, our approach uses the identified set of candidates to find an optimal grouping *G* that minimizes the distance function `dist`, while ensuring that all constraints are met and each activity in A_L is assigned to exactly one group in *G*. We formulate this task as a mixed-integer programming (MIP) problem.

Finally, having obtained a grouping *G*, the third step abstracts the input event log *L* by replacing the events in a trace with activities based on groups defined in *G*, yielding an abstracted log *L'*.

4.3.2 Computation of Candidate Groups

In this first step, our approach computes a set of candidate groups of activities, $\mathcal{G}$, i.e., subsets of A_L that adhere to constraints in *C*. As described in Section 4.3.1, we propose an exhaustive and a DFG-based instantiation for this step:

Exhaustive candidate computation. In order to obtain a complete set of candidate groups, in principle, every combination of activities, i.e., every subset of A_L, needs to be checked against the constraints in *C*. However, we are able to considerably reduce this search space by (for the moment) only looking for candidate groups $g \subseteq A_L$ that actually co-occur in at least one trace of the event log L (referred to as *group co-occurrence*) and by considering the monotonicity of the constraints in *C*. Then, candidate groups of increasing size can be identified in an iterative manner, as outlined in Algorithm 1.

Initialization. We first set a constraint-checking *mode* (line 1) based on the monotonicity of the constraints in *C*; *mode* is set to *anti-monotonic* if *C* contains at least one such constraint, *monotonic* if all constraints in $C \setminus C_G$ are monotonic (i.e., all constraints that must be checked *per group*), and otherwise to *non-monotonic*.

Using this constraint-checking mode, we employ two pruning strategies. First, consider a group $g_1 \subset A_L$ and a constraint set *C* in which all constraints $C \setminus C_G$ are monotonic. If $\texttt{holds}(g_1, C, L) = \mathit{true}$, any supergroup $g_1' \supseteq g_1$ will also adhere to the constraints, since adding more activities to g_1 will never lead to a violation of a monotonic constraint. Therefore, in the *monotonic* mode, we can avoid the costs of constraint validation for g_1'. Second, consider a group $g_2 \subset A_L$, known to

Algorithm 1 Exhaustive candidate computation.

Input Event log L, user constraints C
Output Set of candidate groups $\mathcal{G}$

1: *mode* $\leftarrow$ `setCheckingMode`(C)
2: *toCheck* $\leftarrow \{\ \{c\}$ for $c \in A_L\}$ ▷ Set the first groups to check
3: **while** *toCheck* $\neq \emptyset$ **do**
4: **if** *mode* = *monotonic* **then**
5: $\mathcal{G}_{new} \leftarrow \{g \in$ *toCheck* if $\exists g' \in \mathcal{G} : g' \subset g \vee$ `holds`(g, C, L)
6: **else**
7: $\mathcal{G}_{new} \leftarrow \{g \in$ *toCheck* if `holds`$(g, C, L)\}$
8: $\mathcal{G} \leftarrow \mathcal{G} \cup \mathcal{G}_{new}$
9: **if** *mode* = *anti-monotonic* **then**
10: *toCheck* $\leftarrow$ `expandGroups`($\mathcal{G}_{new}$)
11: **else**
12: *toCheck* $\leftarrow$ `expandGroups`(*toCheck*)
13: *toCheck* $\leftarrow \{g \in$ *toCheck* if `occurs`$(g, L)\}$
14: **return** $\mathcal{G}$

violate any anti-monotonic constraint in C, i.e, `holds`$(g_2, C, L) = \mathit{false}$. Then we also know that no supergroup $g_2' \supseteq g_2$ can adhere to C, as expanding a group can never resolve violations of anti-monotonic constraints. Thus, in the *anti-monotonic* mode, all supergroups of g_2 can be skipped.

With *mode* set, the algorithm adds all activities in A_L as singleton groups to the set of potential candidates, *toCheck*, which are checked in the first iteration (line 2).

Candidate assessment. In each iteration, Algorithm 1 first establishes a set $\mathcal{G}_{new}$, which contains all groups in *toCheck* that adhere to the constraints in C. When validating a group, we check constraints in C_C before ones in C_I, since the former do not require a pass over the event log, thus, minimizing the validation cost per candidate. In the *monotonic* mode, the algorithm employs the first pruning strategy by directly adding any group g, for which there is a $g' \subset g$ already in $\mathcal{G}$, given that we then know that the monotonic constraints will be satisfied for g as well (line 5). For other groups and for the other two modes, we need to check `holds`(g, C, L) for each group $g \in \mathit{toCheck}$ (lines 5 and 7). Having established $\mathcal{G}_{new}$, the new candidates are added to the total set $\mathcal{G}$ (line 8).

Group expansion. Next, the algorithm repopulates *toCheck* with larger groups that are assessed in the next iteration. In the *anti-monotonic* mode, using the second pruning strategy, the algorithm only needs to expand groups that are known to adhere to all anti-monotonic constraints in C. Therefore, in this case, we only expand the groups in $\mathcal{G}_{new}$ (line 10). This expansion involves the creation of new groups that consist of a group $g \in \mathcal{G}_{new}$ with an additional activity from A_L.

Naturally, the *anti-monotonic* mode avoids the creation of groups that contain subgroups that are already known to violate C. For the *monotonic* and *non-monotonic* mode, we also need to expand groups that currently violate the constraints, since their super-groups may still lead to constraint satisfaction. Therefore, these modes expand all groups in *toCheck* (line 12). Afterwards, we only retain those groups in *toCheck*(g, L) that actually occur in the event log, by checking if there is at least one trace in L that contains events corresponding to all activities in g (line 13).

Termination. The algorithm stops if there are no new candidates to be checked, returning the set of all candidates, $\mathcal{G}$.

Computational complexity. Although Algorithm 1 guarantees to find a complete set of candidates, its time complexity is exponential with respect to the number of activities in the event log, i.e., $2^{|A_L|}$. In the worst case, all subsets of A_L must be analyzed against the entire log, where primarily the number of traces is important, since each group must be separately checked against all traces. Given that each checked group may become a candidate, the algorithm's space complexity is also bounded by $2^{|A_L|}$. Hence, this exhaustive approach can quickly become infeasible.

DFG-based candidate computation. Given the run time complexity of the exhaustive approach, we also propose a DFG-based approach to compute candidate groups. It exploits behavioral regularities in event logs in order to derive a set of *cohesive candidate groups* more efficiently than the exhaustive approach.

Intuition. Event abstraction aims to find cohesive groups of activities and, therefore, is more likely to group together activities that typically occur close to each other. In our running example, even though the *request receipt* (*rcp*) and *archive request* (*arv*) activities meet the constraint (both are performed by a *clerk*), it is unlikely that they will end up in the same higher-level activity in an optimal grouping G, since *rcp* occurs at the start of each trace and *arv* at the end.

We exploit this characteristic of optimal groupings by identifying only candidates that occur near each other. This is achieved by establishing a DFG of the event log and traversing this graph to find highly cohesive candidates groups. Since this traversal again iteratively increases the candidate size, we can still apply the aforementioned pruning strategies.

This idea is illustrated in Figure 4.4, which visualizes (parts of) two iterations for the running example, highlighting candidate groups that are checked. Iteration 1 involves the assessment of paths of length two, consisting of connected activities. This identifies, e.g., the candidate paths [*prio,inf*], [*prio,arv*], and [*inf,arv*], which all adhere to the constraint, whereas, e.g., [*acc, inf*] is recognized as a violating path, since *acc* and *inf* are performed by different roles. Given their distance from each other in the DFG, this iteration avoids checking groups such as {*rcp, arv*} and {*ckt, inf*}. In the next iteration, since the example deals with an *anti-monotonic*

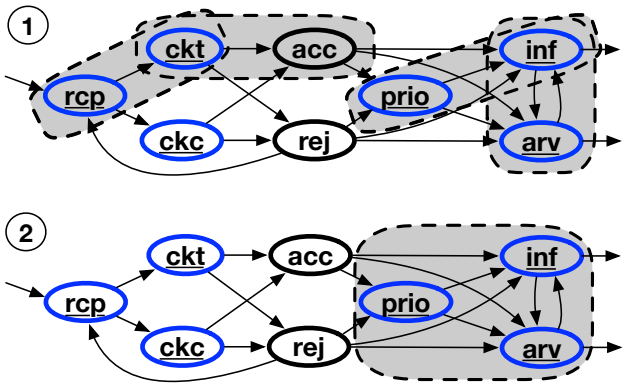


Figure 4.4: DFG-based candidate computation (Iterations 1 & 2).

constraint, we concatenate pairs of constraint-adhering paths to obtain candidate paths (i.e., groups) of length three, as shown for [*prio, inf, arv*] in Figure 4.4.

The DFG-based approach works as described in Algorithm 2. Next to an event log L and a constraint set C, it takes a parameter k, defining the beam-search width.

Initialization. The algorithm starts by again setting the constraint-checking mode (line 1), before establishing the log's DFG as defined in Definition 8(line 2). Then, for every node n in the DFG (i.e., for every activity), we add the trivial path $\langle n \rangle$ to the set of candidates to check in the first iteration (line 3).

Candidate assessment. In principle, we could assess for each path $p \in$ *toCheck* if p's nodes form a proper candidate group, as we do in the exhaustive approach. However, we here recognize that in event logs with a lot of variability, the number of paths to check will still be considerable. Hence, we allow for a further pruning of the search space by incorporating a *beam-search* [181] component in the algorithm. In this beam search, we only keep the k most promising candidates (i.e., the *beam*) in each iteration of the algorithm.

To this end, each iteration starts by sorting the candidate paths in *toCheck*, prioritizing paths of which the nodes have the lowest distance to each other, according to $\texttt{dist}(\texttt{nodes(p)}, L)$ (line 5). Then, the algorithm picks candidates from *sortedPaths* as long as there are candidates to pick and the beam width k has not been reached (line 8). Each group g, defined by a path's nodes (i.e., $g = \texttt{nodes}(p)$), is then checked for constraint satisfaction. As for the exhaustive approach, we check constraints in C_C before ones in C_I minimizing validation cost per candidate.

We use the same pruning strategies for *monotonic* and *anti-monotonic* constraint-checking modes as before. In the *monotonic* mode, a group g can be directly added to the set of candidates $\mathcal{G}$ if we have already seen a subset $g' \subset g$ that adheres to the constraints (lines 12–13), whereas in the *anti-monotonic* mode, we no longer expand paths that violate the constraints (lines 18–19).

Algorithm 2 DFG-based candidate computation.

Input event log L, user constraints C, pruning parameter k
Output Set of candidate groups $\mathcal{G}$

1: *mode* $\leftarrow$ `setCheckingMode(`C`)`
2: $DFG \leftarrow$ `computeDFG(`L`)`
3: *toCheck* $\leftarrow \{\langle n \rangle$ for $n \in DFG$.`nodes`$\}$ ▷ First paths to check
4: **while** *toCheck* $\neq \emptyset$ **do**
5: *sortedPaths* $\leftarrow$ `sort(`*toCheck*, `dist)` ▷ Lowest `dist` first
6: *toExpand* $\leftarrow \emptyset$
7: $i \leftarrow 0$
8: **while** $i < \min(|$*sortedPaths*$|, k)$ **do**
9: $p \leftarrow$ *sortedPaths*$[i]$ ▷ Get next path
10: $g \leftarrow$ `nodes(`p`)` ▷ Derive nodes, i.e., activities of p
11: **if** *mode* = *monotonic* **then**
12: **if** $\exists g' \in \mathcal{G} : g' \subset g \vee$ `holds(`g, C, L`)` **then**
13: $\mathcal{G} \leftarrow \mathcal{G} \cup \{g\}$
14: $toExpand \leftarrow toExpand \cup \{p\}$
15: **else if** `holds(`g, C, L`)` **then**
16: $\mathcal{G} \leftarrow \mathcal{G} \cup \{g\}$
17: $toExpand \leftarrow toExpand \cup \{p\}$
18: **else if** $mode \neq$ *anti-monotonic* **then**
19: $toExpand \leftarrow toExpand \cup \{p\}$
20: $i \leftarrow i + 1$
21: *toCheck* $\leftarrow \emptyset$ ▷ Start computing new paths to check
22: **for** $p = \langle p_0, \ldots, p_m \rangle \in$ *toExpand* **do**
23: **for** $(p_m,$ *succ*$) \in DFG$.`outgoingEdges(`p_m`)` **do**
24: **if** *succ* $\notin$ `nodes(`p`)` **then**
25: *toCheck* $\leftarrow$ *toCheck* $\cup \{ \langle p_0, \ldots, p_m,$ *succ*$\rangle \}$
26: **for** $($*pred*$, p_0) \in DFG$.`incomingEdges(`p_0`)` **do**
27: **if** *pred* $\notin$ `nodes(`p`)` **then**
28: *toCheck* $\leftarrow$ *toCheck* $\cup \{ \langle$*pred*$, p_0, \ldots, p_m \rangle \}$
29: *toCheck* $\leftarrow \{p \in$ *toCheck* if `occurs(nodes(`p`), `L`)`$\}$
30: **return** $\mathcal{G}$

Path expansion. The candidates for the next iteration are created by expanding paths in *toExpand* with either a predecessor of their first or a successor of their last node (lines 22–28). As before, we only retain those paths in *toCheck*, whose groups g actually occur in the event log (line 29).

Termination. Algorithm 2 stops if no candidates remain to be checked, i.e., *toCheck* is empty and the set $\mathcal{G}$ is returned.

Computational complexity. The DFG-based approach is considerably more efficient than the exhaustive one. Each iteration expands up to k groups, each into up to $|A_L| - 1$ new candidates. As such, given the maximum of $|A_L|$ iterations, the worst-case time and space complexity is $k * |A_L|^2$. Moreover, this worst case only occurs if the DFG is a complete digraph and no constraints are imposed.

Dealing with exclusion. Generally, it is undesirable to group exclusive activities together, since these never occur in the same trace. Therefore, we have so far omitted these from consideration, by ensuring that $\texttt{occurs}(g, L)$ holds for every candidate group $g \in \mathcal{G}$. However, exclusive activities (or groups) that are proper alternatives to each other are the exception. Grouping these will result in further complexity reduction of the log, while not affecting its expressiveness.

Intuition. To illustrate this, reconsider the running example, which contains two sets of exclusive activities, {*ckc*, *ckt*}, corresponding to two ways in which a request can be checked, and {*acc*, *rej*}, corresponding to acceptance and rejection of a request. By considering Figure 4.5, we see that the former two activities are proper behavioral alternatives: both *ckc* and *ckt* are preceded and followed by the exact same sets of activities. Therefore, behavioral alternatives can be defined as groups of activities that have identical *pre-* and *postsets* in the DFG. Merging them will thus not lead to a loss of behavioral information. By contrast, *acc* and *rej* do not represent proper behavioral alternatives to each other, since their postsets differ. Particularly, while after acceptance the process always moves forward to one of the activities in {*prio*, *inf*, *arv*}, a rejection may also result in a loop back to the start of the process (*rcp*). Therefore, if these exclusive classes were merged, we would obscure the fact that there are two different possibilities here.

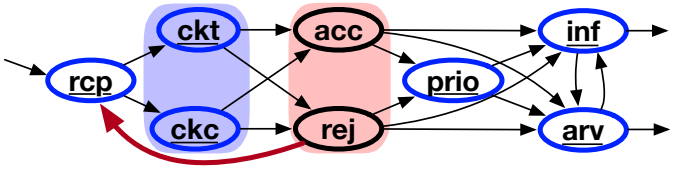


Figure 4.5: DFG of the running example highlighting proper behavioral alternatives (blue) and exclusive activities, which are no behavioral alternatives (red).

Candidate identification. Algorithm 3 determines if previously identified candidates in $\mathcal{G}$, with excluding activities, can be merged to obtain additional candidates.

The algorithm establishes a set *equivGroups* consisting of candidate groups that share the same pre- and postset (line 4). Then, a stack is created consisting of all pairs of groups in this set (lines 5–7). For each pair (g_i, g_j) in this stack, we assess if g_i and g_j are indeed exclusive to each other and if their merged group,

g_{ij}, still adheres to the user constraints (line 11). Both conditions can be efficiently checked. The former by ensuring that there are no edges from nodes in g_i to nodes in g_j or vice versa, while for the latter only adherence to class-based constraints (C_C) needs to be assessed, given that instance-based constraints cannot be (newly) violated when merging exclusive groups, thus avoiding a pass over the event log L.

If g_{ij} is indeed a proper, new candidate, we next determine if this group can also be combined together with its preset, postset, or with both, to create more candidates (lines 13–19). For instance, having identified {*ckt, ckc*} as a new candidate group for the running example, we would this way recognize that this new group together with its preset (*rcp*) also forms a proper candidate group: {*rcp, ckt, ckc*}, since both {*rcp, ckt*} and {*rcp, ckc*} were also already part of $\mathcal{G}$.

Having established these new candidates, Algorithm 3 adds any new pair (g_{ij}, g_k) to the stack, so that also iteratively larger candidates, comprising three or more exclusive groups, can be identified (lines 20–21). It terminates when all relevant pairs have been assessed, returning $\mathcal{G}$ as the final output of this step.

Computational complexity. Algorithm 3 has linear complexity with respect to $|\mathcal{G}|$, i.e., the number of candidate groups stemming from the previous step. As such, its worst-case time and space complexity is $2^{|A_L|}$ when previously using the exhaustive approach and $k * |A_L|^2$ for the DFG-based one.

4.3.3 Finding an Optimal Grouping

Having established candidate groups $\mathcal{G}$, we set out to find an optimal grouping $G \subseteq \mathcal{G}$ based on these candidates, which is a set of disjoint groups that covers all activities, while minimizing the overall distance. We formulate this task as a MIP problem, which can be tackled using standard solvers.

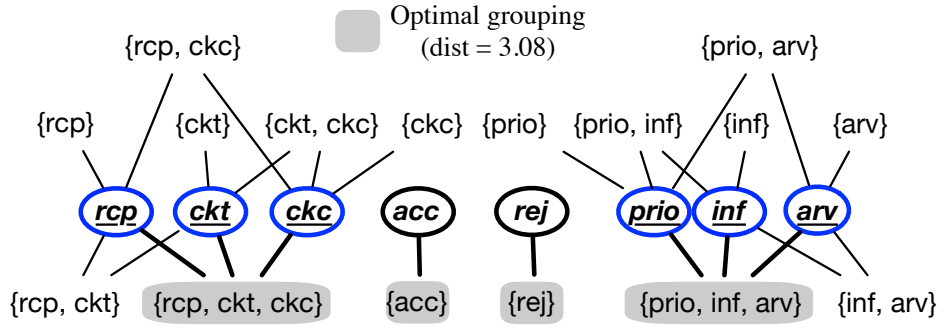


Figure 4.6: Optimal grouping of the running example given all candidates computed in the first step using the DFG-based approach.

Algorithm 3 Finding exclusive candidate groups.

Input Event log L, user constraints C, current candidate groups $\mathcal{G}$
Output Extended set of candidate groups $\mathcal{G}$

$DFG \leftarrow \texttt{computeDFG}(L)$
seenGroups $\leftarrow \emptyset$
for $g \in \mathcal{G} \setminus$ *seenGroups* **do**
    equivGroups $\leftarrow DFG.\texttt{equalPrePost}(g) \cup \{g\}$
    pairsToCheck $\leftarrow$ `new Stack()`
    for $(g_i, g_j) \in$ *equivGroups* $\times$ *equivGroups* **do**
        pairsToCheck.push$((g_i, g_j))$
    while $\neg$*pairsToCheck*.isEmpty() **do**
        $(g_i, g_j) \leftarrow$ *pairsToCheck*.pop()
        $g_{ij} \leftarrow g_i \cup g_j$ ▷ Merge into single group
        if $\texttt{exclusive}(g_i, g_j) \wedge \texttt{holds}(g_{ij}, L, R_C)$ **then**
            $\mathcal{G} \leftarrow \mathcal{G} \cup \{g_{ij}\}$ ▷ New candidate found
            (*pre, post*) $\leftarrow (DFG.\texttt{pre}(g_i), DFG.\texttt{post}(g_i))$
            if *pre* $\cup$ *post* $\cup\, g_i \in \mathcal{G} \wedge$ *pre* $\cup$ *post* $\cup\, g_j \in \mathcal{G}$ **then**
                $\mathcal{G} \leftarrow \mathcal{G} \cup \{$*pre* $\cup$ *post* $\cup\, g_{ij}\}$
            else if *pre* $\cup\, g_i \in \mathcal{G} \wedge$ *pre* $\cup\, g_j \in \mathcal{G}$ **then**
                $\mathcal{G} \leftarrow \mathcal{G} \cup \{$*pre* $\cup\, g_{ij}\}$
            else if *post* $\cup\, g_i \in \mathcal{G} \;\wedge\;$ *post* $\cup\, g_j \in \mathcal{G}$ **then**
                $\mathcal{G} \leftarrow \mathcal{G} \cup \{$*post* $\cup\, g_{ij}\}$
            for $g_k \in$ *equivGroups* $\setminus \{g_i, g_j\}$ **do**
                pairsToCheck.push$((g_{ij}, g_k))$
            equivGroups $\leftarrow$ *equivGroups* $\cup \{g_{ij}\}$
    seenGroups $\leftarrow$ *seenGroups* $\cup$ *equivGroups*
return $\mathcal{G}$

This formulation is based on a bipartite graph $(\mathcal{G}, A_L, E)$, which connects each candidate group to the activities it covers, i.e., it contains an edge $(g_i, a_j) \in E$ if $a_j \in g_i$. Figure 4.6 visualizes this for the running example, in which the circled nodes in the middle indicate activities in A_L, the sets indicate the candidate groups $\mathcal{G}$, and the edges their coverage relation. The grayed sets highlight the optimal grouping in this case, which is an exact cover because every event-class node is connected to exactly one of the selected groups. Given this bipartite graph $(\mathcal{G}, A_L, E)$, we formalize a MIP problem with two decision variables:

- $selected_{g_i} \in \{0, 1\}$: 1 if $g_i \in \mathcal{G}$ is selected, else 0;
- $covered_{a_j} \in \{0, 1\}$: 1 if $a_j \in A_L$ is covered, else 0.

We aim to minimize the distance of the selected groups in the objective function:

$$\text{minimize} \sum_{g_i \in \mathcal{G}} \texttt{dist}(g_i) * selected_{g_i}$$

This objective function is subject to two constraints:

$$\sum_{j=1}^{|A_L|} covered_{a_j} = |A_L| \tag{4.3}$$

$$\sum_{(g_i,a_j)\in E} selected_{g_i} = covered_{a_j}, \forall a_j \in A_L \tag{4.4}$$

These constraints jointly express that each activity must be covered (Equation 4.3), by exactly one group (Equation 4.4). In case a user imposes grouping constraints in C_G, bounding the number of groups that may be selected, these are imposed by adding either or both of the following additional constraints:

$$\sum_{i=1}^{|\mathcal{G}|} selected_{g_i} \leq x \quad \text{resp.} \quad \sum_{i=1}^{|\mathcal{G}|} selected_{g_i} \geq y \tag{4.5}$$

The selected groups, i.e., $G = \{g_i \in \mathcal{G} : selected_{g_i} = 1\}$, then form the obtained grouping. Note that, depending on the characteristics of log L and the imposed constraints C, a grouping may not be found, since there is no guarantee that a feasible solution exists. In that case, our approach returns the initial log. To allow users to then refine their constraints appropriately, our approach also indicates possible causes of infeasibility, e.g., the affected activities that lead to violations for constraints in C_C, or the fraction of cases for which constraints in C_I are violated. If a solution is found, our approach continues with its third and final step.

Computational complexity. Most MIP problems are NP-hard, although an assessment of the exact complexity depends on the concrete problem and is poorly characterized by input size [162]. However, solvers like *Gurobi* often solve MIP problems efficiently, e.g., by applying heuristics. Our experiments confirm this, showing that Step 2 only contributes marginally to the overall run time of our approach.

4.3.4 Creating an Abstracted Event Log

Finally, we use the grouping G to establish abstracted versions of the traces in L to establish the output of our approach, an abstracted event log L'.

For each trace $\sigma \in L$, we identify all activity instances in the trace, i.e., all instances of groups in G, $\mathcal{I}^{\sigma} = \bigcup_{g \in G} \texttt{inst}(\sigma, g)$. Each activity instance, $\xi_i \in \mathcal{I}^{\sigma}$, corresponds to an ordered sequence of events, $\langle e_1, ..., e_k \rangle$.

Next, our approach creates an abstracted trace σ', which reflects the activity instances in $\mathcal{I}^{\sigma}$, instead of the events of its original counterpart, σ. A common abstraction strategy is to let σ' capture only the *completion* of activity instances, by creating a projection of σ that only retains the last event, e_k, per activity instance. For instance, for trace $\sigma_1 = \langle$rcp, ckc, acc, prio, inf, arv$\rangle$ of the running example, this abstraction would yield $\sigma_1^c = \langle$clrk1, acc, clrk2$\rangle$.

Yet, this strategy may obscure information when activities are executed in an interleaving manner. For this, consider a new trace $\sigma_5 = \langle$rcp, ckc, prio, acc, inf, arv$\rangle$. Here, events belonging to the *clrk2* group occur both before (*prio*) and after (*inf, arv*) the unary activity instance *acc*. When only retaining completion events, this yields the trace $\sigma_{5'}^c = \langle$clrk1, acc, clrk2$\rangle$, which hides the interleaving nature of the activities. Therefore, we also propose an alternative strategy, which retains both the *start (s)* and *completion (c)* events per activity instance $\xi_i \in \mathcal{I}^{\sigma}$. This yields a trace $\sigma_{5'}^{s+c} = \langle clrk1_s, clrk1_c, clrk2_s,$ acc, $clrk2_c \rangle$, which thus shows that activity *clrk2* starts before *acc* and completes afterwards. The choice for a particular strategy depends on the relevance of parallelism in a particular analysis context, given that the latter strategy also leads to longer traces, thus partially mitigating the benefits of the obtained abstraction.

The log L' that results from this last step represents the final output of our approach. It is an event log in which the high-level activities are guaranteed to satisfy the user-defined constraints in *C* while providing a maximal degree of abstraction.

4.4 Evaluation

To evaluate our approach, we conducted experiments using a collection of real-world event logs, which we describe in Section 4.4.1. Section 4.4.2 outlines the evaluation setup. Section 4.4.3 reports on the results obtained for our approach and its configurations and compares our work against three baselines. Finally, Section 4.4.4 further illustrates the value of constraint-driven log abstraction through two application cases. The employed implementation, evaluation pipelines, and additional experimental results are all publicly available.[1]

[1] `https://github.com/a-rebmann/constraint-driven-abstraction`

4.4.1 Data Collection

We use a collection of 13 publicly-available event logs. To be able to cover various constraints, the events in all of these logs have at least one categorical event attribute, as well as timestamps used for numerical constraints. Table 4.3 shows they vary considerably in terms of key characteristics, e.g., the number of activities.

Table 4.3: Properties of the real-world log collection.

Ref.	$\|A_L\|$	**Traces**	**Variants**	$\|E_L\|$	**Avg.** $\|\sigma\|$
[53]	11	150,370	231	70	3.73
[69]	40	75,928	3,453	357	6.35
[65]	39	46,616	22,632	772	10.01
[67]	24	31,509	5,946	180	16.41
[68]	39	14,550	8,627	407	52.48
[64]	24	13,087	4,366	125	20.04
[62]	8	10,035	1	14	15.00
[70]	51	7,065	1,478	553	12.25
[169]	4	1,487	183	10	4.47
[38]	27	1,434	116	99	5.98
[124]	16	1,050	846	115	14.49
[66]	70	902	295	124	24.00
[132]	29	20	20	164	69.70

4.4.2 Evaluation Setup

We conducted our evaluation experiments based on the following setup.

Implementation and environment. We implemented our approach in Python, using *PM4Py* [32] for event-log handling and *Gurobi* [90] as a solver for MIP problems. All experiments were conducted single-threaded on an Intel Xeon 2.6 GHz processor with up to 768GB of RAM available.

Constraints. We use ten constraint sets, covering the various constraint types that our approach supports. Each set includes the class-based constraint $|g| \leq 8$, which is used to limit the number of abstraction problems that time out. This constraint is combined with each of the sets from Table 4.4, covering anti-monotonic (AM), monotonic (MO), and non-monotonic (NM) instance-based constraints, a grouping constraint (GR), as well as two sets of their combinations ($C1$ and $C2$). Table 4.4 also contains additional constraints ($BL1$ to $BL4$) used in baseline comparisons, described below. By combining these constraint sets with the 13 event logs, we establish a total of 121 abstraction problems to be solved.[2]

[2] $BL3$ can only be applied to 4 out of 13 logs, as the others lack class-level event attributes.

Table 4.4: Constraints used in the experiments.

ID	Categories	Constraint(s)
AM	C_I	$\lvert g.role\rvert \leq 3$
MO	C_I	$\texttt{sum}(g.duration) \geq 10^1$
NM	C_I	$\texttt{avg}(g.duration) \leq 5 * 10^5$
GR	C_G	$\lvert G\rvert \leq 3$
$C1$	C_I, C_G	$AM \wedge NM \wedge GR$
$C2$	C_I, C_G	$AM \wedge MO \wedge NM \wedge GR$
$BL1$	C_C	$\lvert g\rvert \leq 5$
$BL2$	C_C	$BL1 \wedge \texttt{cannotLink}(e_1.a, e_2.a)$
$BL3$	C_C	$\lvert g.d\rvert = 1$
$BL4$	C_G	$\lvert G\rvert = {}^{\lvert L\rvert}/_2$

Configurations. We test three configurations that differ in the instantiation of the first step of our approach (cf. Section 4.3.2):

- **Exh**, using exhaustive candidate computation;
- **DFG**$_\infty$, using the DFG-based instantiation without beam search (i.e., unlimited beam width);
- **DFG**$_k$, using the DFG-based instantiation with a beam width that adapts to the number of activities in the given log, i.e., $k = 5 * |A_L|$.

Note that we let candidate computation time out after 5 hours. our approach then continues with the candidates identified so far.

Baselines. We compare our approach against three baselines. These represent alternative approaches to satisfy the optimal-event-abstraction task (Definition 10) and differ in the scope of constraints they can handle.[3]

*Graph querying (***BL**$_Q$*).* Our approach's DFG-based candidate computation traverses a DFG to find candidate groups that adhere to imposed constraints. Recognizing the overlap of this with graph querying, BL$_Q$ replaces Step 1 of our approach with an instantiation using graph querying. For this, the DFG is stored in a graph database, which is queried for candidate groups using constraints formulated in a state-of-the-art graph querying language [82]. Given that a DFG captures a log on the class-level (i.e., on the level of activities), BL$_Q$ can only support class-based constraints, though. Thus, we assess BL$_Q$ using a constraint on the maximum group size ($BL1$), an additional cannot-link constraint between activities ($BL2$), and a constraint over a class-level attribute ($BL3$). By comparing against BL$_Q$, we aim to show that our approach yields more comprehensive sets of candidate groups than those obtained by adopting existing solutions.

[3] Additional details on the baselines and their implementation are available through our repository.

*Graph partitioning (**BL**$_P$).* Our approach's goal to find a disjoint set of cohesive groups for log abstraction is similar to the goal of graph partitioning, which aims to partition a graph such that edges between different groups have a low weight [179]. Therefore, we compare our approach against a baseline using such partitioning, BL$_P$. Given a DFG, BL$_P$ aims to minimize the sum of directly-follows frequencies of cut edges, while cutting the graph into n partitions. For this, BL$_P$ applies *spectral partitioning* [179], where the weighted adjacency matrix is populated using normalized directly-follows frequencies. Since graph partitioning simply splits a DFG into a certain number of groups, BL$_P$ can only support strict grouping constraints, whereas instance-based, class-based, and flexible grouping constraints (e.g., constraint GR), cannot be handled. Therefore, we compare BL$_P$ against our approach using the constraint $BL4$, which aims to reduce number of activities of a log by half. This comparison aims to show that our approach leads to better abstraction results, while also supporting a considerably broader range of constraints.

*Greedy approach (**BL**$_G$).* Finally, we compare our approach against a greedy event abstraction strategy. BL$_G$ starts by assigning all activities from A_L to a set of singleton groups, G_0. Then, in each iteration, BL$_G$ merges those two groups from G_i that lead to the lowest overall distance, i.e., $\texttt{dist}(G_{i+1}, L)$, without resulting in any constraint violations. BL$_G$ stops if the overall distance cannot improve in an iteration. Unlike the other baselines, BL$_G$ can handle instance-based constraints, since it works directly on the event log rather than the DFG, although, grouping constraints cannot be enforced in this iterative strategy. Therefore, we compare BL$_G$ against our approach using the instance-based constraint set AM, MO, and NM. This comparison against BL$_G$ aims to show the importance of striving for a global optimum in the log-abstraction problem.

Measures. To assess the results obtained by the various configurations and baselines, we consider the following measures:

*Solved abstraction problems (**Solved**).*: We report on the fraction of solved problems, to reflect the general feasibility of abstraction problems and the ability of a specific configuration to find such feasible solutions.

*Size reduction (**S. red.**).*: We measure the obtained size reduction by comparing the number of high-level activities in an obtained grouping to the number of original activities, i.e., $|G|/|A_L|$. Given the strong link between model size and process understandability [159], this measure provides a straightforward but clear quantification of the abstraction degree.

*Complexity reduction (**C. red.**).*: We also assess the abstraction degree through the reduction in *control-flow complexity*, using an established complexity measure [159]. Since this measure requires a process model as input, we discover a model for both the original and the abstracted log using the state-of-the-art *Split Miner* [24] and then compare their complexity.

*Silhouette coefficient (**Sil.**).*: We quantify the intra-group cohesion and inter-group separation of a grouping G using the *silhouette coefficient* [102], an established measure for cluster quality. To avoid bias, we compute this coefficient using a standard measure for the pair-wise distance between activities [89], which considers their average positional distance.

*Run time (**T(m)**).*: Finally, we measure the time required to obtain a result in minutes, from the moment a log L is imported until the abstracted log L' is returned.

4.4.3 Evaluation Results

In this section, we report on the results for the different constraint sets, followed by a comparison of the different configurations.

Overall results. Table 4.5 presents the results obtained using the Exh configuration of our approach per constraint set. For the anti-monotonic (AM, BL1-3) and grouping constraint sets (GR,$BL4$) our approach finds a solution to all of the problems. Infeasible problems primarily occur for the monotonic MO constraint set and the combination sets, $C1$ and $C2$, since these are more restrictive. Interestingly, $C1$ has more than twice as many solved problems (54%) than $C2$ (23%), clearly showing the impact of $C2$'s additional monotonic constraint on feasibility.

The other measures in the table report on the results obtained for the solved problems. We observe that our approach achieves a considerable degree of abstraction, reflected in the reductions in size and complexity. Groupings are reasonably cohesive and well separated from each other, indicated by silhouette coefficients $\geq$ 0.12. These results are stable for the less restrictive constraint sets, such as AM, NM, and GR, as well as their combination $C1$. For instance, for AM a size reduction of 0.68, complexity reduction of 0.63, and silhouette coefficient of 0.15 is achieved. In line with expectations, for more restrictive constraint sets, e.g., *C2*, the impact of abstraction is less significant (0.50, 0.40, and 0.09 resp.). Finally, the impact of the constraint-checking modes on efficiency can also be observed.[4] While, e.g., the GR constraint set requires 144m on average to be solved, the anti-monotonic $BL2$ constraint cases are solved in 121m. In this mode, candidates are not expanded if they already violate the constraint, yielding improved run times.

Overall, our approach is thus able to greatly reduce the size and complexity of event logs, while respecting various constraints. Although the solution feasibility and the abstraction degree depends on the employed constraints, our approach consistently finds groups that have strong cohesion and good separation.

[4]For constraint sets with unsolved problems, run times must be compared carefully, as they depend on the specific logs with feasible solutions.

Table 4.5: Results for Exh (averaged over solved problems).

Const.	Solved	S. red.	C. red.	Sil.	T(m)
AM	1.00	0.68	0.63	0.15	146
MO	0.31	0.58	0.55	0.15	75
NM	0.77	0.68	0.65	0.12	154
GR	1.00	0.66	0.61	0.13	144
$C1$	0.54	0.68	0.59	0.12	134
$C2$	0.23	0.50	0.40	0.09	100
$BL1$	1.00	0.67	0.61	0.12	122
$BL2$	1.00	0.66	0.61	0.12	121
$BL3$	1.00	0.38	0.29	-0.02	38
$BL4$	1.00	0.51	0.46	0.05	147

Exhaustive versus efficient configurations. Table 4.6 depicts the evaluation results for the three our approach configurations, again providing the averages over the solved problems. Notably, the configurations were able to solve the same problems, except for a single problem in the non-monotonic NM constraint set, which the DFG$_k$ configuration failed to solve. We observe that the DFG-based configurations achieve substantial efficiency gains in comparison to the exhaustive one, where in particular DFG$_k$ needs only about 40% of the time in comparison to Exh (49m vs. 130m on average). With respect to the abstraction degree, we observe that DFG$_\infty$ maintains results comparable to Exh for size (0.62 vs. 0.63) and complexity reduction (0.56 vs. 0.57). It even obtains better results for the silhouette coefficient (0.16 vs. 0.11), which shows the ability of the DFG-based approach to find candidate groups that are cohesive and well-separated. The results achieved by DFG$_k$ suggest a trade-off between optimal abstraction and efficiency, as the abstraction degree is about 7% lower compared to the other configurations. Finally, we observe that the DFG-based configurations are particularly useful for anti-monotonic and grouping constraints. In these cases, the results differ only marginally, even for DFG$_k$, while achieving considerable efficiency gains.

Table 4.6: Results per configuration over solved problems.

Conf.	Solved	S. red.	C. red.	Sil.	T(m)
Exh	0.78	0.63	0.57	0.11	130
DFG$_\infty$	0.78	0.62	0.56	0.16	108
DFG$_k$	0.77	0.56	0.50	0.08	49

Baseline Results

Table 4.7 depicts the results obtained using the baseline approaches against the most relevant configurations of our approach.

Comparison to graph querying. The results of BL_Q indicate that the candidate groups obtained using graph queries are not as comprehensive as those found by our approach's DFG_∞ configuration. BL_Q's solutions are therefore subpar with respect to size and complexity reduction. Furthermore, the negative silhouette coefficients (-0.2 avg. vs. 0.17 for DFG_∞) indicate that the groupings found by BL_Q are neither cohesive nor separated, which highlights the ability of DFG-based candidate computation to find better sets of candidates for abstraction.

Comparison to graph partitioning. With respect to BL_P we find that partitioning the DFG by minimizing edge cuts naturally reduces the size of the DFG and, thus, achieves a certain degree of abstraction. However, the groupings created by BL_P are not as cohesive, indicated by the silhouette coefficient of 0.01 compared to our approach (0.05). Moreover, the complexity reduction achieved by BL_P (0.41) is lower than achieved by our approach (0.46), even though their groupings contain the same number of activities. This highlights the benefits of the three-step approach our approach takes and the suitability of its distance measure to obtain meaningful groupings for log abstraction.

Comparison to greedy approach. When considering the results of BL_G, the downsides of a greedy solution strategy quickly become apparent. BL_G finds solutions to fewer abstraction problems (64%) than even the most efficient configuration, DFG_k, whereas the solutions that are identified are far subpar. For example, for the anti-monotonic AM constraint set, BL_G achieves an average size reduction of 0.47, whereas DFG_k yields a size reduction of 0.64, which clearly shows that a greedy strategy often yields solutions that are far from optimal.

Table 4.7: Baseline comparison over the applicable constraint sets. Results are averaged over solved problems.

Const.	Conf.	Solved	S. red.	C. red.	Sil.	T(m)
BL[1–3]	**DFG**$_\infty$	1.00	0.63	0.55	0.17	77
	BL$_Q$	0.96	0.55	0.43	-0.20	24
$BL4$	**Exh**	1.00	0.51	0.46	0.05	147
	BL$_P$	1.00	0.51	0.42	0.01	1
AM, MO, NM	**DFG**$_k$	0.67	0.59	0.52	0.08	58
	BL$_G$	0.64	0.45	0.37	0.02	24

Discussion. Overall, these results demonstrate that our approach outperforms all three baselines with respect to their applicable constraints, whereas it can, furthermore, handle a much broader range of process-oriented abstraction constraints.

4.4.4 Application Cases

In this section, we apply our approach in two application cases to give an illustration of the value of constraint-driven event abstraction.

To this end, we use the BPI17 log [67] that captures a loan application process at a large financial institution. Although the log only contains 24 activities, its complexity is considerable, as evidenced by its 160 DFG edges. As shown in Figure 4.7 this issue even remains for a so-called 80/20 model, which omits the 20% least frequent edges, since this still provides few useful insights into the underlying process. Clearly, there is a need for abstraction, for which our approach can be applied.

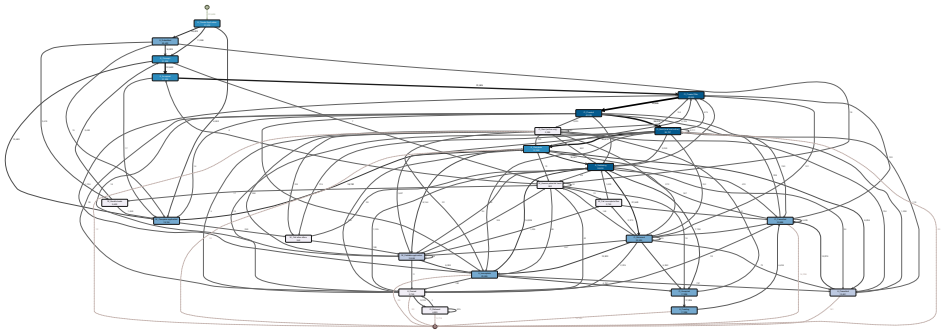


Figure 4.7: 80/20 DFG discovered from the BPI17 event log that captures a loan application process [67].

Origin-aware abstraction. We recognize that the needed abstraction can be guided by considering the three IT systems from which events in the log originate: an *application-handling* system (A), the *offer* system (O), and a general *workflow* system (W). Since these systems each relate to a distinct part of a process, we impose a constraint that avoids mixing up events from different systems into a single activity, i.e., $|g.origin| \leq 1$.

Figure 4.8 depicts the DFG obtained by our approach in this manner, where the activity labels reflect their origin systems. Having grouped the events into seven high-level activities, the DFG shows a considerable reduction in terms of size and complexity. Due to this simplification, we observe clear inter-relations between the different sub-systems. For instance, the process most often starts with the execution

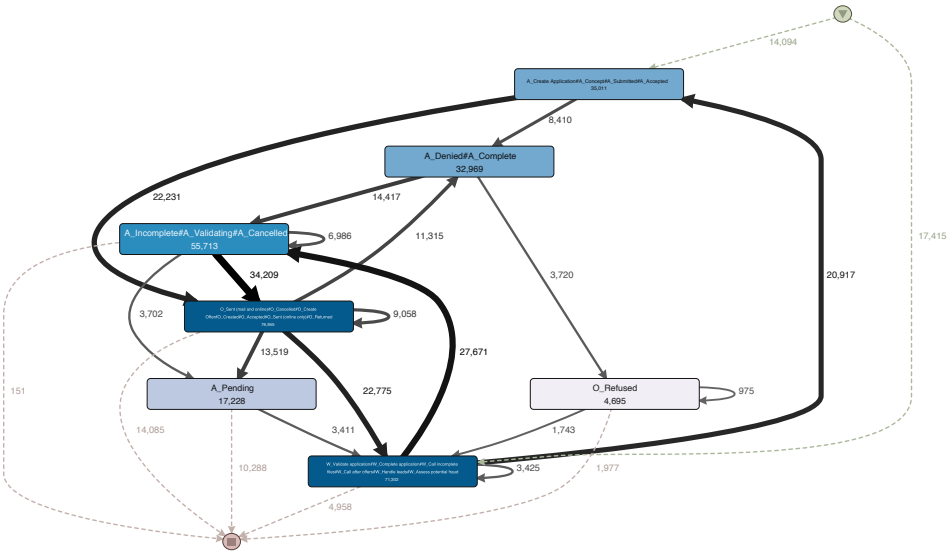


Figure 4.8: 80/20 DFG of the abstracted loan application log.

of steps in the application-handling system (*A_Activity 1* to *3*), followed by a part in the offer system (*O_Activity 1*), and again concluded in the application-handling system (*A_Activity 4*). Next to this main sequence, the workflow-related steps (*W_Activity 1*) occur in parallel to the other activities, whereas the refusal of an offer (*O_Activity 2*) represents a clear alternative path.

Semantics-aware event abstraction. Next, we show how the identified components and categorized actions that become available by applying our annotation approach (Chapter 3) can be used for semantics-aware event abstraction.

We observe that the events in the loan-application log primarily relate to actions performed for two main types of objects: *applications* and *offers*. Moreover, while various actions are applied to these objects, their action type reveals that these primarily relate to *create*, *modify*, and *decide* actions. We use these insights to establish two constraints: (1) grouped activities should refer to the same object type and (2) they should refer to the same type of action. We apply our abstraction approach using these constraints. We then name the obtained high-level activities as a combination of their action type and object type, resulting in the abstracted DFG of Figure 4.9. The graph contains the 80% most frequent edges, while omitting activities that do not refer to either of the main types of objects.

Having obtained this abstracted view on the process reveals clear dependencies between action types within and across the two object types. For instance, we find that applications are first created and processed (modified). Moreover, a decision

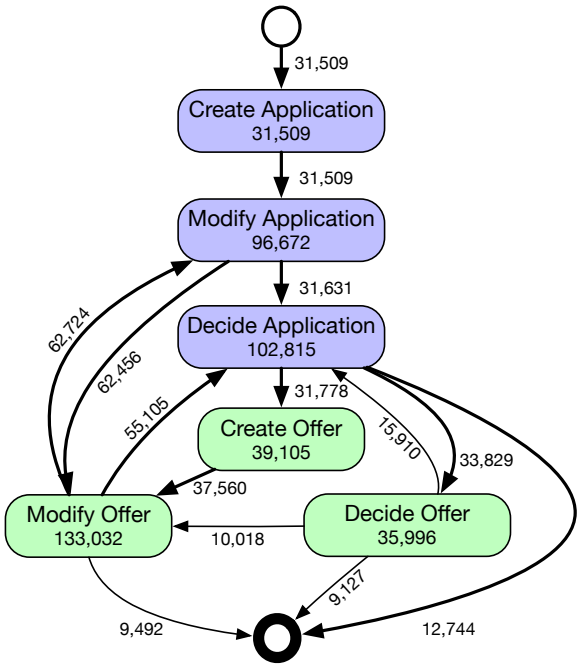


Figure 4.9: Abstracted 80/20 DFG that shows the actions types applied to the main type of objects in the log, if any.

about an application, i.e., its acceptance, is necessary before an offer can be created. Further, there can be multiple iterations between offer and application. After modifying the offer, the application can be modified before the offer is modified again. Finally, the majority of cases end with a decision, either about an offer or about the application. Another common end of the process is a final modification of the offer. By looking into the original activities, this corresponds to sending a notification about an offer.

It is important to stress that the insights of these two application cases are only possible due to the constraint-driven nature of our work. In fact, when applying our approach without imposing any constraints, the intertwined nature of the process yielded high-level activities that contain events from all three sub-systems respectively both object types, thus obfuscating the key inter-relations in the process.

4.5 Limitations

Our experiments reveal that our approach is capable of effectively abstracting event logs, while guaranteeing user-defined characteristics in the output log. However, in its current form our work has certain limitations, which relate to the approach and the evaluation.

The scope of our approach is currently limited to constraints on individual groups, such that, e.g., instance-based constraints over the entire grouping (rather than per group) are currently not supported, which may be relevant for certain analysis purposes. Second, the complexity of the task at hand results in our approach to exhibit long run times (even when using the more efficient heuristic version) for event logs with a considerable amount of different activities and traces. Therefore, further research is needed to improve this performance by, for instance, using trace-sampling techniques to obtain approximate solutions. Finally, while our approach effectively groups events of an input event log, it cannot automatically generate semantically meaningful labels for the higher-level activities. Instead, it requires a user to manually assign such labels based on the groups' content.

As for our evaluation, we acknowledge that the quantitative experiments and application cases provide limited insights into the efficacy of our approach in real-world scenarios. To address this limitation, user studies could be conducted that investigate the usefulness of our approach in supporting users with different analysis purposes that require event abstraction.

4.6 Related Work

The work on constraint-driven event abstraction that this chapter presents primarily relates to event abstraction in process mining, behavioral pattern mining from event logs, and sequential pattern mining.

4.6.1 Event Abstraction

A broad range of event-abstraction approaches has been proposed by the process mining community, each with its own requirements and assumptions. In their event-abstraction taxonomy, Van Zelst et al. [187] consider the degree of *supervision*, i.e., the amount of information on how to abstract that an abstraction approach requires as input, as a main differantiator. This is also the main dimension considered by Diba et al. [60] in their review on event abstraction.

The spectrum of information that different abstraction approaches require is shown in Figure 4.10. Fully unsupervised approaches mostly employ clustering techniques [81, 158] or generic abstraction patterns [99, 180] to map low-level

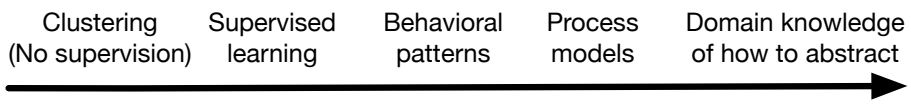


Figure 4.10: Spectrum of information on how to abstract required by different event-abstraction approaches. Adopted with slight modifications from [60].

events to higher-level ones. Supervised approaches often require users to provide information about the high-level activities to be discovered, captured, e.g., in the form of behavioral patterns, process models [26], specific event annotations [106], or even detailed domain knowledge, such as event hierarchies [104].

Our approach enables a user to impose requirements on the abstracted event log such that it serves a particular analysis purpose. Unlike our approach, existing approaches do not guarantee any characteristics for the abstracted logs, which can result in a considerable loss of information. At the same time, our approach only needs a specification of *what* properties a user requires, whereas existing supervised approaches require a user to explicitly specify *how* abstraction should be performed. As such, it can be positioned on the left of the spectrum, between clustering and supervised learning.

4.6.2 Behavioral Pattern Mining from Event Logs

Behavioral pattern mining also lifts low-level events to a higher degree of abstraction by identifying interesting patterns in event logs, including constructs such as exclusion, loops, and concurrency. Local Process Models (LPMs) [172] provide an established foundation for this. LPMs are mined according to pattern frequency, while extensions have been proposed to employ interest-aware utility functions [171] and incorporate specific types of user constraints [173]. Behavioral pattern mining has been also addressed through the discovery of maximal and compact patterns in event logs [16] including a context-aware extension [15]. While their purpose is similar, a key difference between behavioral pattern mining and event abstraction is that the former "cherry-picks" interesting parts from a log, whereas the latter strives for comprehensive abstraction over an entire event log.

4.6.3 Sequential Pattern Mining

Approaches for sequential pattern mining [20, 139] identify interesting patterns in sequential data. As for LPMs, *interesting* is typically defined as *frequent* [93], while techniques for *high-utility sequential pattern mining* also support utility functions specific to the data attributes of events [175, 184, 185]. Cohesion, com-

parable to distance measures used in event abstraction, has also been applied as a utility measure for pattern mining in single long sequences [49, 50]. Furthermore, research in *constrained sequential pattern mining* primarily focuses on exploiting constraint characteristics such as monotonicity to improve efficiency [140], which we leverage during the first step of our approach. Often the focus is on specific constraints, e.g., time-based gap constraints [22]. In contrast to our approach, pattern mining techniques do not consider concurrency and exclusion. Moreover, like for behavioral pattern mining, the focus is on the identification of individual patterns, while our work strives for global abstraction.

4.7 Summary

This chapter proposed an approach for constraint-driven abstraction of event logs. With our approach, users can define the desired characteristics and requirements of abstracted logs in terms of constraints, thus enabling the meaningful and purpose-driven abstraction of low-level event data. We defined the optimal-event-abstraction task that requires minimizing the distance to the original log while satisfying a set of constraints on the abstracted log. To satisfy this task, our approach covers a broad set of common constraint types and a distance measure. We provided two primary instantiations of our approach, allowing users to trade-off computational efficiency and result optimality.

Our quantitative evaluation experiments on 13 real-world event logs reveal that our work considerably outperforms three baseline techniques. Furthermore, we present two application cases that demonstrate the usefulness of constraint-driven abstraction in practical settings.

Chapter 5

Task-Level-Event Recognition from User Interaction Data

User interaction data comprises events that capture individual actions that a user performs on their computer. Each user interaction event corresponds to a single interaction between the user and the user interface of a software application, such as clicking a button, entering a text into a field, or ticking a checkbox [13, 109]. Therefore, such events provide detailed records about how users carry out their activities in a process (which we, here, refer to as *tasks*). In process mining settings, user interaction data provides a lot of potential, since it records events across applications, at a detailed level, and without the need to extract or integrate data from different source systems.

However, user interaction events are unsuitable to be directly used for process mining, as they do not meet two essential requirements. First, user interaction events do not indicate their relation to a process-level activity. Consequently, when analyzing a process using such events, insights will show how a user interacted with their applications, rather than how the process was executed. For instance, in the context of an order-handling process, applying process mining to user interaction events would result in insights such as *input text* is commonly followed by *click button*, instead of insights such as *create order* is commonly followed by *update order*. Second, user interaction events do not relate to specific process executions, which means that the relation between different process steps is not captured. For example, having identified a number of process steps that involve the handling of orders, it is crucial to understand which of these steps relate to the same customer order and which to different ones. Hence, to enable process mining based on user interaction data, it must be transformed so that it meets the requirements of events in process mining settings (which we, here, call *task-level events*).

In this chapter, we propose an unsupervised approach for recognizing task-level events that addresses this problem. It segments user interaction data to identify tasks, categorizes these according to their type, and relates tasks to each other via object instances it extracts from the low-level events. In this manner, our approach creates task-level events that can be used in process mining settings. In addition to being the first approach that provides an end-to-end solution to this transformation problem, it can also deal with online streams of user interaction events. It emits task-level events to a stream that can directly be used as input for streaming process mining techniques [39]. These, in turn, can provide a timely understanding of current process behavior and enable process monitoring and predictions on-the-fly.

This chapter is based on a journal paper titled "*Recognizing Task-level Events from User Interaction Data*" [150] by Adrian Rebmann and Han van der Aa, which is an extension of the conference paper "*Unsupervised Task Recognition from User Interaction Streams*" [151].

The remainder of this chapter is structured as follows. Section 5.1 illustrates the challenges of recognizing task-level events from user interaction data. Section 5.2 presents our approach, which we evaluate in Section 5.3. Section 5.4 reflects on limitations of our work. Finally, Section 5.5 discusses related work and Section 5.6 provides a summary.

5.1 Problem Illustration

In this section, we first describe the two main parts involved in recognizing task-level events from user interaction data, before highlighting the additional challenges of doing this in a streaming setting.

Recognizing task-level events. To illustrate the problem of recognizing task-level events in an unsupervised manner, consider the excerpt of event data in Table 5.1, where the events record how a user handles requests related to orders. Although the user interaction events show what a user does at a detailed level, it fails to give clear information about the actual process that is executed. In particular, the user interaction events neither make their relation to process-relevant tasks nor to specific process executions explicit. For instance, they do not indicate that events *u1–u8* correspond to the execution of a particular task, i.e., creating order *O007501*, and events *u9–u16* to a different one, i.e., updating order *O008102* after a change request was made. Identifying these relations involves the following two parts:

1. *Identify tasks and their types.* The first part involves identifying groups of user interaction events that jointly form tasks and their types. This can be achieved in a two-step manner:
   (1) We need to find sequences of user interaction events that together form individual tasks. Working under the assumption that a user performs one

Table 5.1: Excerpt of a user interaction data recording two task executions.

ID	Action	Application	Timestamp	Element	Label	Value
...	...	...	...	...	...	...
u1	click	Mail	15:41:32	list	Order	-
u2	input	Chrome	15:42:10	field	Login	-
u3	input	Chrome	15:42:26	field	Password	-
u4	click	Chrome	15:42:31	button	ok	-
u5	click	Chrome	15:43:01	button	Create order	O007501
u6	input	Chrome	15:43:29	field	Search	Pete Miller
u7	input	Chrome	15:43:43	field	Customer	C0075
u8	click	Chrome	15:43:58	button	Save order	O007501
u9	click	Mail	15:44:32	list	Change request	-
u10	input	Chrome	15:44:41	field	Customer	C0081
u11	click	Chrome	15:45:39	button	Edit order	O008102
u12	input	Chrome	15:45:48	field	Quantity	4
u13	click	Chrome	15:46:05	button	Save order	O008102
u14	click	Chrome	15:46:39	button	Edit order	O008102
u15	input	Chrome	15:46:48	field	Quantity	5
u16	click	Chrome	15:46:55	button	Save order	O008102
...	...	...	...	...	...	...

task before moving to the next, this involves the identification of points in the data where one task completes and the next one starts. In the given example, this is the case after events *u8* and *u16*, which denote the completion of two higher-level tasks. The difficulty here is that such end points are not explicitly indicated in the data. For instance, although *u8* ends the first task by the press of a *Save order* button, event *u13* involves such a button as well, even though it occurs only halfway through the execution of the second task. As such, this requires identifying when execution has moved to the next task, based on clues from the context and attributes of events.
(2) Having identified individual tasks, we aim to recognize which tasks correspond to the same type (e.g., creating an order), and which to different ones. However, variability makes this difficult, since the same process-level task may be executed by performing different sequences of user interaction events. For instance, the *create order* task (*u1–u8*) could also be executed without first logging in (*u2–u4*) or by having to search multiple times (*u6*) until the right customer is found.

2. *Identify task relations.* The second part is to identify the relations of tasks to process executions. To identify such relations, process-relevant object instances, such as specific *orders* and *customers*, provide valuable information.

For example, by identifying object instances in the events of the running example, we can recognize that the events comprising the first task (*u1–u8*) do not relate to those comprising the subsequent one (*u19–u16*), since they, respectively, relate to orders *O007501* and *O008102*. However, extracting such information is challenging, because user interaction events generally record object instances only implicitly. In particular, there are typically no dedicated attributes associated with user interactions that capture information about process-related object instances. Instead, these are spread across attributes that describe user interface elements and their values, such as button labels or input values. For example, event *u5* does not make its relation to *order O007501* explicit. Instead, the object type, *order*, is contained in its `Label` and the identifier (*O007501*) in its `Value` attribute.

This problem is partially addressed in robotic process mining (RPM) [109], where the main purpose is to identify automatable tasks in user interaction data. For instance, the RPM approach by Leno et al. [108] identifies tasks by segmenting user interaction data. However, it neither recognizes a task's type nor its relation to a process execution, so that the first part is only partially solved, while the second part is not addressed at all. The approach by Urabe et al. [177] goes beyond segmentation and also clusters identified tasks, thus addressing the first part of the transformation problem, yet also leaving the second part unaddressed.

Therefore, we propose an approach for recognizing task-level events that addresses both parts. Furthermore, unlike existing approaches, ours works in streaming settings. This comes with additional challenges, though, as explained next.

Challenges of the streaming setting. Our work (also) targets online settings, in which user interaction events arrive in a stream. Recognizing task-level events from a stream is more complex than doing it in an offline manner using an event log, due to the general constraints of streaming settings [34]. Specifically, we have to identify tasks, recognize their type, and extract objects to infer their relations as they are observed, using just a limited buffer to temporarily store a relatively small number of events. This leads to two main difficulties:

1. *Single-pass processing.* In an offline setting, approaches can do multiple passes over an entire collection of events, allowing them to use global information, such as overall co-occurrence counts [177], when making decisions. However, in a streaming setting, events cannot be accessed indefinitely [39], so that decisions have to be made on the basis of potentially incomplete information, e.g., the counts observed up to a certain point in the stream.
2. *Adapting to changes over time.* An associated issue is that when dealing with streams, decisions have to be made without knowing what kind of events and objects will arrive in the future. For example, while offline approaches can be certain that all types of tasks they need to identify are already available,

this is not the case in a streaming setting. At any point in time, events corresponding to new kinds of applications, actions, objects, or task types may be observed. For instance, for the running example, events *u9–u16* must be properly analyzed, even if no such *update order* task has been seen before, which requires on-the-fly updating of the recognition mechanisms.

5.2 Recognition Approach

In this section, we describe our approach for recognizing task-level events from user interaction data. We first give an overview of its input, components, and output, before describing the individual components in detail.

5.2.1 Approach Overview

Input. Our approach takes as input low-level event data in the form of a user interaction stream. In line with the definitions of Leno et al. [108], we define user interaction events, user interaction classes, and user interaction streams (see also Definition 6) as follows.

Definition 11 (User interactions and user interaction events) *A* user interaction *is a manual action performed on a user interface, such as clicking a button or entering a value into a text field. We denote a* user interaction event *(*UI-event*)* $u = (uid, ts, P, V)$ *as a tuple that records a user interaction, with* $\mathcal{U}$ *the universe of all UI-events. Each UI-event has a unique identifier* $u.uid$*, a timestamp* $u.ts$*, a set of context attribute values* $u.P$*, capturing the interaction type and information about the affected user interface element, and a set of data attribute values* $u.V$*, capturing data associated with an interaction, e.g., what the user typed into a field.*

For instance, *u6* = (*u6*, *15:43:29*,{*input*, *Chrome*, *field*, *Search*}, {*Pete Miller*}).

Definition 12 (UI-classes) *Given a UI-event, we let its context attributes values, i.e.,* $u.P$*, define its UI-class. We use the shorthand* $X.P$ *to refer to the set of UI-classes of all UI-events in a collection* X*, with* $X \subset \mathcal{U}$*.*

For instance, the UI-class of *u6* is given as {*input*, *Chrome*, *field*, *Search*}.

Definition 13 (User interaction streams) *A user interaction stream* S_U *is a potentially infinite sequence of UI-events recorded during task execution, i.e.,* $S_U \in \mathcal{U}^* \forall_{1 \leq i < j \leq |S_U|} S_U(i) \neq S_U(j)$*.*

Approach components. Figure 5.1 provides a high-level overview of our approach, which we complement with a formalization in Algorithm 4. As depicted, it recognizes task-level events from a user interaction stream S_U based on three components: the *object-instance-identification component* determines to which object instances UI-events relate, the *task-identification component* identifies sequences of UI-events that correspond to individual tasks, and the *task-categorization component* assigns a type to an identified task.

While object-instance identification is applied to each UI-event as soon as it arrives, the task-identification and categorization components operate on the basis of an event buffer B. We assume that B is large enough to store the UI-events comprising a single task instance. In online settings, our approach applies task categorization as soon as it identified a task based on the UI-events in the buffer. In offline settings, it first applies task identification to an entire UI-event log and only then continues with task categorization.

Output. For each recognized task, our approach emits a start and a completion event to a task-level event stream S_T (cf. Definition 6). In line with Definition 1, each task-level event $t \in T \subset \mathcal{E}$ consists of a task's type (i.e., activity, $t.a$), its timestamp ($t.ts$), and the following data attributes: its object instances ($t.objects$), a task identifier ($t.id$), and lifecycle information ($t.lifecycle$) that indicates whether it corresponds to the task's start or completion.

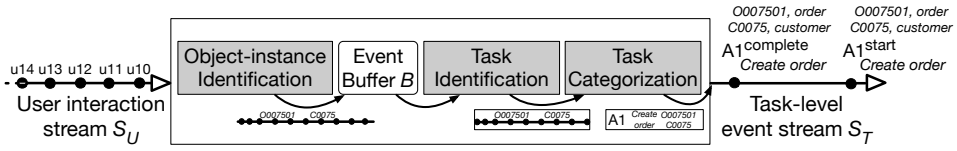


Figure 5.1: Overview of the recognition approach.

5.2.2 Object-Instance Identification

First, our approach identifies object instances in the UI-events, which (1) it uses in the subsequent components and (2) which indicate if and how recognized task-level events relate to each other through shared object instances (line 4). As shown in Figure 5.2, the object-instance-identification component consists of two parts that are applied for each UI-event u: First, *type extraction* aims to extract an object type ot from u. Then, *instance recognition* detects if u indeed refers to an instance of ot and—if so—adds it to the current task's object instances. In the following, we explain these parts in detail.

Algorithm 4 Recognizing task-level events from user interaction data.

Input S_U: User interaction stream, b: Maximum buffer size
Output S_T: Task-level event stream

▷ Initialize buffer B, clustering model CM, chunk list C, and object instance set $objects$
$B \leftarrow$ new FIFOQueue(b), $CM \leftarrow$ new ClusteringModel(), $C \leftarrow []$, $objects \leftarrow \emptyset$
loop forever
  $u \leftarrow S_U$.observeEvent() ▷ A new UI-event is consumed from the stream
  ▷ Object-instance identification
  $objects \leftarrow objects \cup \{(u.uid, \text{identifyObject}(u))\}$
  ▷ Add the UI-event to the buffer
  B.insert(u)
  ▷ Task identification
  if completesChunk(u) **then**
    C.add(B.getEventsSinceLastChunk(C)) ▷ Create and store new chunk
  if $|C| \geq 2$ **then** ▷ Check if enough chunks available
    $c_i, c_{i+1} \leftarrow C[-2], C[-1]$ ▷ Get chunks to be checked
    if endsTask(B, C, c_i, c_{i+1}, $objects$) **then**
      $taskEvents \leftarrow (B$.dequeueUpThrough(c_i)) ▷ De-queue UI-events
      $C \leftarrow C$.removeRange($C[0], c_i$) ▷ Remove chunks
      ▷ Task categorization
      $v \leftarrow$ vectorize($taskEvents, objects$) ▷ Create a feature vector
      CM.update(v) ▷ Update the clustering model
      $type \leftarrow CM$.categorizeTask(v) ▷ Assign task type
      $label \leftarrow$ label, getDefiningTerms($taskEvents$) ▷ Assign task label
      ▷ Emit task-level events
      $tid \leftarrow$ newID() ▷ Assign an ID to the task
      $taskObjects \leftarrow \{obj \mid (uid, obj) \in objects\}$
      $objects \leftarrow \emptyset$ ▷ Empty the objects set for the next task
      emit(S_T, (tid, $type$, $taskEvents[0].ts$, {(lifecycle, *start*), (label, $label$)}, $taskObjects$))
      emit(S_T, (tid, $type$, $taskEvents[-1].ts$, {(lifecycle, *complete*), (label, $label$)}, $taskObjects$))}

Type Extraction

The first part of this component extracts an object type ot from a given UI-event u. For instance, it aims to detect that *u1*, *u5*, and *u8* each refer to an *order*, while *u7* refers to a *customer*. This involves *noun identification* and *UI-object removal*.

Noun identification. Recognizing that object type information is often contained in context attribute values, e.g., in button labels such as *Save order*, noun identification establishes a set of nouns N_u from the context attribute values P of u.

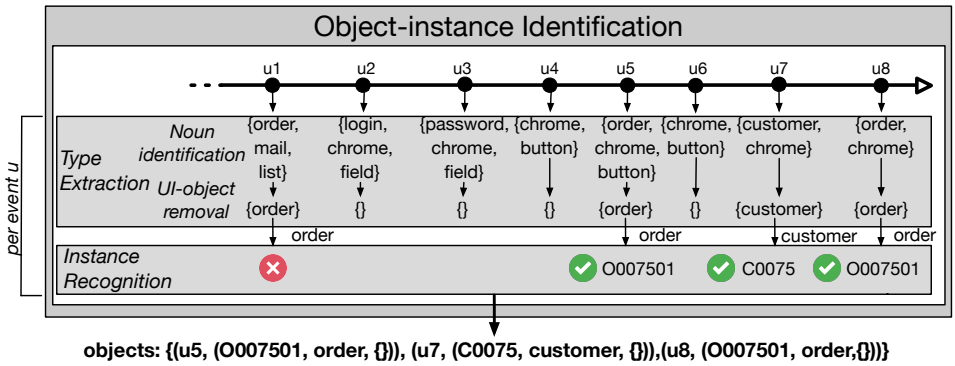


Figure 5.2: Object-instance-identification component.

To establish N_u, we employ a POS-tagger provided by standard NLP tools (e.g., *spaCy* [97]). Given a context-attribute value p, e.g., *Create order* (see *u5*), the tagger assigns linguistic roles to individual words in p, e.g., it assigns VERB to *Create* and NOUN to *order*. Using this, we instantiate a function nouns that, given a value $p \in u.P$ returns the set of nouns in p. For instance, nouns(*Create order*) $= \{order\}$. Noun identification applies nouns to all $p \in u.P$, which results in a set $N_u = \bigcup_{p \in u.P} \text{nouns}(p)$ that may contain process-related nouns. Yet, it likely also contains nouns that rather relate to the user interface itself, which we remove next.

UI-object removal. Having identified a set of nouns N_u, UI-object removal discards any nouns that pertain to user interface elements (e.g., *textfield* or *excel*) rather than process-related objects, as these cannot be used to establish meaningful relations between tasks.

To this end, we use a set K_U of user-interface-specific terms, which consists of names of different UI elements, such as *button*, *field*, and *link*, common application names, such as names of browsers, spreadsheet applications, text-processing software, productivity tools, and application-specific objects, such as *workbook*, *sheet*, and *cell* in case of MS Excel[1]. If a noun $n \in N_u$ corresponds to a term in K_U, it is not process-related and thus removed from N_u. In this manner, our approach establishes a set of process-related nouns $N_u^{ot} = N_u \setminus K_U$.

Output. If N_u^{ot} still contains nouns after this removal step, i.e., $N_u^{ot} \neq \emptyset$, our approach concatenates these to represent the object type ot of u. For instance, $N_u^{ot} = \{order\}$ becomes $ot = order$, while $N_u^{ot} = \{order, line\}$ becomes $ot = order\ line$ before it continues with instance recognition for u. If no nouns remain, it continues with type extraction for the next UI-event.

[1]For the full set K_U of terms we refer to our repository linked in Section 5.3

Instance Recognition

Having extracted an object type ot from u, the object-instance identification component next aims to recognize if u indeed refers to a specific instance of ot. For example, it recognizes that the *O007501* `Value` of UI-event *u5* represents the identifier of the specific order that was created by *u5*.

To this end, instance recognition establishes a set of identifying values I_u^{oi} that jointly represent an object identifier oi based on u's value attributes V. We recognize that, in the context of UI-events, such identifying values are typically alphanumeric IDs, URLs, email addresses, and names of people, organizations, and places. Therefore, our approach adds any value $v \in u.V$ to I_u^{oi} that (partially) consists of digits, corresponds to a URL or email address, or refers to a named entity such as *Pete Miller* (see *u6*). Note that digits, URLs, and email addresses can be straightforwardly detected using regular expressions, whereas named entities, i.e., persons, organizations, countries, cities, etc., can be detected using NER capabilities of standard NLP tools [97].

Instance recognition then concatenates any identifying values in the same manner as done for nouns in type extraction to represent the object identifier oi.

Provided that the object-instance-identification component extracts an object type and recognizes a corresponding identifier in u, it establishes an object instance $o = (oi, ot, \{\})$ according to Definition 4.[2] Finally, the component adds the new instance to the set of current objects (line 4). Regardless of whether an object instance was added or not, the component then continues with the next UI-event.

5.2.3 Task Identification

The task-identification component identifies sequences of UI-events from the stream that correspond to individual tasks. It consists of two main operations, as visualized in Figure 5.3. Here, *chunking* identifies sequences of observed UI-events that represent sub-tasks, such as filling in a form or sending an e-mail, whereas *segmenting* determines if consecutive sub-tasks corresponds to the same process-level task or rather to different ones. Once such a transition from one task to the next has been detected, we forward the segment that corresponds to the completed task to our task-categorization component.

[2]Note that the empty map represents the object's value map, which may later be used to associate the object with specific properties, such as monetary amounts in case of orders.

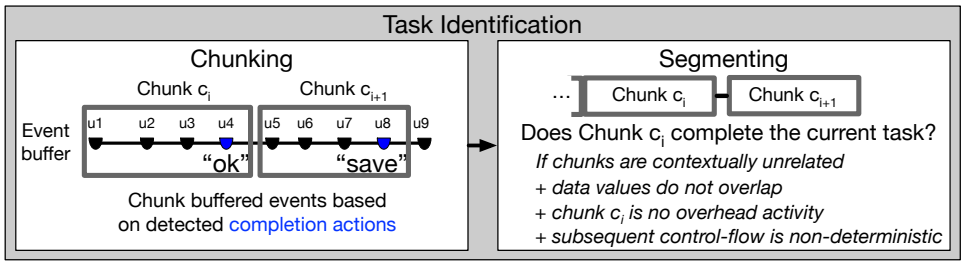


Figure 5.3: Task-identification component.

Chunking

We recognize sub-tasks by looking for common keywords in user interaction data that indicate the conclusion of an interaction sequence, achieved through the `completesChunk` function in Algorithm 4 (line 6). To operationalize this function, we use a set of completion actions K_A, which consists of 20 keywords stemming from design guidelines for user interfaces by IBM [98], covering typical terms that indicate the conclusion of a smaller part in a process, such as *ok* (to go to the next step in a user interface), *submit* (for a form), *send* (e-mail), or *save* (changes).[3]

For a UI-event u, `completesChunk`(u) returns true if u's class contains a mention of an action in K_A. Based on the UI-events stored in B, a sub-task is formed by the UI-events that occurred since the last completed chunk in C (line 7). For instance, for the running example, *u4*, *u8*, *u13*, and *u16* complete chunks (due to their *ok* and *save* labels), resulting in *u1–u4*, *u5–u8*, *u9–u13*, and *u14–u16* as chunks.

Segmenting

The segmenting operation aims to decide whether a chunk c_i corresponds to the end of a task or if it continues with the next chunk, c_{i+1} (function `endsTask` in line 10). Specifically, as shown in Figure 5.3, `endsTask` identifies c_i as finalizing a task if: (1) the chunks are contextually unrelated to each other, (2) the chunks have no overlap in data values, (3) c_i does not represent an overhead activity, and (4) the control-flow after c_i is non-deterministic. Otherwise, c_i and c_{i+1} are considered to belong to the same task.

(1) Assessing contextual relatedness. Our approach first checks if c_i and c_{i+1} are contextually related or not. We do this by lifting the notion of contextual relatedness proposed by Urabe et al. [177], which targets offline segmentation, to our setting. The idea is to check if the UI-classes contained in c_i and c_{i+1} commonly

[3] We refer to our repository for the full list of keywords, though K_A can naturally be extended with, e.g., self-defined keywords or other languages.

co-occurred so far (indicating a shared context) or not (suggesting that the chunks belong to different tasks).

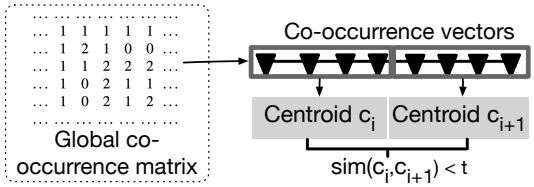


Figure 5.4: Contextual-relatedness approach inspired by Urabe et al. [177].

As illustrated in Figure 5.4, contextual relatedness is quantified on the basis of a global co-occurrence matrix, which tracks how often pairs of UI-classes have been observed to be part of the same chunk. Based on the global counts, we obtain the co-occurrence vectors of the UI-classes per chunk (i.e., rows in the co-occurrence matrix) and compute their centroid. Then, we compute the similarity score $\texttt{sim}(c_i, c_{i+1})$ as the cosine similarity between the centroids of c_i and c_{i+1}. Given a similarity threshold $t \in [0, 1]$[4], we consider the two chunks contextually unrelated if $\texttt{sim}(c_i, c_{i+1}) < t$.

In this manner, given the four chunks identified in the previous operation, we would determine that the transitions from *u1–u4* (logging in) to *u5–u8* (creating an order), and from *u5–u8* (creating an order) to *u9–u13* (updating a quantity) are both clear changes in context. By contrast, the transition from *u9–u13* to *u14–u16* occurs within the same context (updating and fixing an order quantity), due to the chunks' strongly related UI-classes.

(2) Checking for data value overlap. Next, we recognize that sub-tasks may be part of the same task, even when they relate to different contexts, such as opening a request sent by a customer per e-mail and subsequently updating one of their orders in a system. Therefore, we check if UI-events belonging to chunks c_i and c_{i+1} share particular attribute values, including IDs of identified object instances, such as customer names or order numbers. Specifically, we check the last two UI-events of c_i and the first two of c_{i+1} for exact matches in their attribute sets V, and, if such matches are present, determine that there should be no segmentation between c_i and c_{i+1}.

In this manner, we would, for instance, recognize that chunks *u9–u13* and *u14–u16* also relate to each other in terms of their data values, because UI-events *u13* and *u14* both refer to order *O008102*, thus avoiding segmentation here.

(3) Checking for overhead sub-tasks. Then, we check if c_i actually corresponds to a sub-task performed for a particular process instance or that it, rather, corresponds to overhead being performed. Common examples of this include logging

[4] t is configurable and we set it to 0.3 by default.

into a system, launching an application, or visiting non-work related websites. If c_i represents such an overhead sub-task, we do not want to treat it as a distinct task on a process level, which is why we would not segment after c_i (even though contextual relatedness or shared data values between c_i and c_{i+1} are unlikely).

To operationalize this check, we established a set K_O of overhead keywords based on the guidelines [98] we also use for chunking, including *log in*, *sign up*, *reload*, and *open*. Using this set, we check if a member of the last two UI-classes of c_i is contained in K_O and, if so, avoid segmentation. In this manner, we, e.g., recognize that the first sub-task in our running example (*u1–u4*), which corresponds to a user logging into a web app, belongs to the same task as the next chunk *u5–u8*, where the same app is used to create an order.

Note that if one wants to specifically investigate the occurrence of overhead actions in a process, one can simply disable this check in our approach, which will result in such chunks being treated as separate tasks.

(4) Checking for control-flow determinism. Finally, we check if chunks that consist of c_i's UI-classes have always been followed by the same behavior so far. Such control-flow determinism suggests that c_i does not complete a task because it is always necessary to perform the exact same steps after it.

To be able to check for control-flow determinism, we count how often sets of UI-classes directly follow each other throughout the stream and compare it to the number of times the set of c_i's UI-classes, $c_i.P$, formed a chunk so far. If these counts are the same, i.e., $\texttt{count}(c_i.P) = \texttt{countDF}((c_i.P, c_{i+1}.P))$[5], the chunk's subsequent control-flow is considered to be deterministic.[6] This suggests that their corresponding sub-tasks belong to the same task and we avoid segmentation after c_i. These counts can be stored efficiently by identifying UI-classes through their index in the co-occurrence matrix used in the first check, thus, only storing sets of indices and their counts instead of storing sets of entire UI-classes.

Note that the checks based on contextual relatedness and control-flow determinism may benefit from a *warm-up phase*, during which we populate the co-occurrence matrix and control-flow counts for a certain number of UI-events before making the first segmentation decision based on them.

Post-Processing

When our approach has detected that c_i represents the final chunk of a task (line 10), this means that all UI-events currently in the buffer, up to and including the final UI-event of c_i, together form a task. The UI-events that comprise the

[5] `DF` stands for directly-follows.

[6] We only apply this check if $\texttt{count}(c_i.P) \geq 3$, to avoid using it for new sets of UI-classes.

task are then forwarded to the task-categorization component and removed from buffer B as well as chunk list C (lines 11–12), so that the first UI-event in B is the first UI-event of the next task.

5.2.4 Task Categorization

The task-categorization component assigns a type to identified tasks. Given that we cannot store identified tasks in a streaming setting, we categorize them directly after task identification. This is complex, though, because it means we may not yet have observed all possible task types.

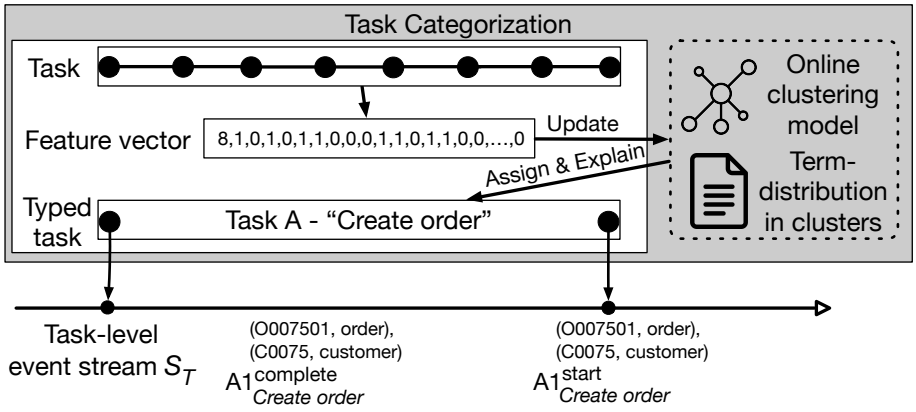


Figure 5.5: Task-categorization component.

To deal with this challenge, we perform task categorization on the basis of an online clustering model CM, which is incrementally updated as new task instances arrive. As shown in Figure 5.5, this involves the transformation of a task into a feature vector, updating the model CM, then using it to assign a cluster to the task, and—finally—providing a textual label for the task.

Establishing Feature Vectors of Tasks

Given an identified task, we first transform its contents into a feature vector that can be used for clustering (line 13). We use a vector encoding that accounts for variability in the executions of tasks of the same type, such as tasks that consist of slightly different sets of UI-classes or that are performed in a different order. Therefore, we capture the number of unique UI-classes (as an indicator of a task's complexity), and the frequency of each UI-class (to capture its contents) as fea-

tures, with a fixed position for each UI-class.[7] For instance, the task *u1–u8* in our running example consists of eight UI-classes that are all performed once, resulting in a vector $\langle 8, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, \ldots, 0\rangle$. Here, the zeros at the end are used to ensure that vectors remain of the same size s_v throughout the stream[8], accounting for a number of UI-classes not seen so far. In offline settings, the vector size can be set to the number of unique UI-classes in the entire input log plus one (to account for the number of unique UI-classes in the task).

Clustering Tasks

We use an online clustering model CM to recognize tasks that are of the same type, based on their vector representation. Specifically, we use *DenStream* [40], a density-based online clustering technique building on the DBSCAN algorithm [78]. It dynamically creates, updates, and deletes so-called micro-clusters in the online feature space it maintains. This technique has two key benefits. First, it is highly memory efficient, since it only stores summaries of vector sets (the micro-clusters), rather than the vectors themselves. Second, unlike many other clustering techniques, it does not depend on a user-defined number of desired clusters.

As shown in Figure 5.5 and Algorithm 4, we update CM with the vector v that corresponds to an identified task (line 14), before using it to assign the task to a cluster (line 15). For instance, due to the distinct features of the two tasks in our running example, these are assigned to different clusters (e.g., types *A* and *B)*.

To improve the performance of task categorization, it can also be beneficial to introduce a *warm-up phase*, which postpones assignment of task types to a point in time when the clustering model has already been updated with a number of identified tasks, and thus is more mature than it is with a cold start. Note that, in offline settings, such a warm-up phase effectively spans the entire event log, since we can then establish a clustering model based on all identified tasks, before assigning categories to them.

Task Labeling

After using clustering to recognize the type of a task, we provide the task with a textual label that indicates what its type actually means.

We automatically generate a suitable label for a given task, by considering the terms that are distinctive to the UI-classes for tasks in its assigned cluster. For this,

[7] Note that the encoding could also capture the types of object instances identified in the UI-events that comprise a task, but in our experiments this did not lead to performance improvements.

[8] Given that, in online settings, the final number of UI-classes is unknown, s_v should be set sufficiently large. We set 1,000 as the default for s_v, which already covers more than 6 times the total number of UI-classes in our evaluation data.

we use the well-known *term frequency–inverse document frequency* (tf-idf) score. Specifically, we use a term dictionary to keep track of the frequency of terms used in the UI-classes within tasks of a specific type, i.e., cluster. Using $CM.types$ to refer to the types currently recognized by the clustering model CM, we write $\texttt{tf-idf}(x, type)$ as the score for a term x and a type $type \in M.types$.

Then, we set the label of *type* as the term x with the highest $\texttt{tf-idf}(x, type)$, e.g., *Create order* for type *A*. If multiple types are assigned the same label (e.g., if *Quantity* is the most distinctive term for types *B* and *C*), we add the term with the next highest `tf-idf` score to each label, until they are all unique. For example, after basic post-processing (term re-ordering and removing capitalization), *B* may get the label *Edit order*, while *C* gets *Confirm quantity*.

5.2.5 Output

The output of our approach is a stream of task-level events based on the identified and categorized tasks as well as the object instances identified in the UI-events that comprise the tasks. For each recognized task, we emit a start event with the timestamp of its first UI-event and a completion event with the last timestamp (lines 20–21). Both task-level events are assigned the task's type (along with its label), a task identifier (created in line 17), and its object instances. For example, for *u1–u8*, we emit:

te_1= {*A* : *Create order*, *15:41:32*,{(`ID`, *1*), (`lifecycle`, *start*),
(`objects`, {(C0075, customer, {}), (O007501, order, {})})}}

te_2= {*A* : *Create order*, *15:43:58*,{(`ID`, *1*), (`lifecycle`, *complete*),
(`objects`, {(C0075, customer, {}), (O007501, order, {})})}}

and for *u9–u16* we emit:

te_3= {*B* : *Edit order*, *15:44:32*,{(`ID`, *2*), (`lifecycle`, *start*),
(`objects`, {(C0081, customer, {}), (O008102, order, {})})}}

te_4= {*B* : *Edit order*, *15:46:55*,{(`ID`, *2*), (`lifecycle`, *complete*),
(`objects`, {(C0081, customer, {}), (O008102, order, {})})}}

Compared to the UI-events in our running example, our approach thus emits events that relate to a process-level activity and to a specific process execution and, thus, fulfill the requirements of process mining settings.

5.3 Evaluation

In this section, we describe the evaluation experiments we conducted. We describe the data collection used in our experiments in Section 5.3.1 and the setup

in Section 5.3.2. In Section 5.3.3 we present the evaluation results showing our approach's capability to automatically recognize task-level events from user interaction data in comparison to baseline approaches. The implementation, data collection, evaluation pipeline, and raw results can be found in our repository.[9]

5.3.1 Data Collection

We aim to show that our approach is capable of recognizing task-level events of varying types. However, there are no publicly available event logs (let alone streams) that contain interaction data related to different task types that are associated with a necessary gold standard. Therefore, we follow the idea of Urabe et al. [177] and take available task logs, each recording various instances of the same type, and combine these into three evaluation logs, which thus cover multiple task types, in various orders. We use these event logs as input in offline settings and as a basis to simulate event streams in online settings.

Task Logs

As shown in Table 5.2, we have eight task logs from three sources available, which include gold-standard task instances (though contained task IDs) and associated types (through their description in the source).[10]

Table 5.2: Characteristics of the task logs used in our experiments.

Task Log	Description	#Tasks	Avg.len.	#Var.	#Events	#Classes
1 [108]	Copy data	100	14.5	7	1,462	15
2 [108]	Reimbursement	50	62.3	1	3,113	32
3 [108]	Student record	50	30.8	2	1,539	23
4 [18]	Travel request	40	73.5	2	2,940	48
5 [13]	New struct. unit	30	10.9	30	331	9
6 [13]	New chapter	30	10.1	30	425	12
7 [13]	New org. unit	30	14.2	30	304	8
8 [13]	New specification	30	11.0	30	326	9

The types of tasks that these logs cover can be divided into two groups:

[9] `https://github.com/a-rebmann/task-recognition`

[10] Note that these logs do not include a gold standard for object instances. To establish that gold standard, the author of this thesis and another researcher annotated user interaction events with correct object instances independently. Afterwards, these were compared and any discrepancies were settled in a discussion. The gold standards are available in our repository.

1. Task logs 1–4 involve *copying address data* from spreadsheets to web forms, entering *reimbursement details*, entering *student records* into a web-based app, and filling in *travel requests*. Logs 1–3 were all published by Leno et al. [108], whereas log 4 stems from a tutorial given by Agostinelli et al. [18]. Note that task log 1 originally contained substantially more instances (1,000 versus 50) and UI-events (14,582 versus 1,539–3,113) than the others. Therefore, we use a sample of its tasks, so that we maintain a more balanced distribution among types as well UI-events per log. To this end, we randomly select 100 instances using stratified sampling with respect to the different execution variants.
2. Task logs 5–8 all relate to the creation of informational objects, such as organizational units, in an SAP system and are provided by Abb and Rehse [13].

Four out of these eight task logs (1–4) contain overhead tasks such as logging into a system, starting an application, or opening a file. As shown in the table, the logs also differ considerably in their variation and task lengths. Note that we also unified the data structure across the task logs as much as possible before combining them, such that attribute names are the same for all task types.

User Interaction Logs

We established three evaluation logs (L_{U1}–L_{U3}) by combining the tasks from individual task logs:
***L_{U1}*:** L_{U1} consists of instances from task logs 1–3 (which are also used by Urabe et al. [177] in their evaluation). We start from an empty log L_{U1} and (1) randomly select a task log and an instance from it, (2) add this instance to L_{U1}, and (3) remove it from the task log, until all instances have been added to L_{U1}. The resulting user interaction log L_{U1} consists of 200 tasks and has a total of 6,114 UI-events.
***L_{U2}*:** We established L_{U2} from task logs 1–4, in the same way as L_{U1}. The resulting user interaction log L_{U2} consists of 240 tasks and contains 9,054 UI-events.
***L_{U3}*:** L_{U3} is based on task logs 5–8. We use these task logs separately from the other ones, since their event attributes, and thus UI-classes, differ considerably. In particular, none of the UI-classes that logs 5–8 contain, occur in logs 1–4. Combining them into one evaluation log would, thus, considerably simplify the identification of transition points between tasks. We combined logs 5–8 in the same way as done for L_{U1} and L_{U2}, so that the resulting user interaction log L_{U3} consists of 120 tasks and contains 1,386 UI-events.

We provide all logs (task logs 1–8 and L_{U1}–L_{U3}) in our repository alongside the implementation. When performing experiments for online settings, we use L_{U1}–L_{U3} to simulate UI-streams (S_{U1}–S_{U3}).

5.3.2 Setup

This section describes the environment, configurations, baselines, and measures used in our experiments.

Environment

We implemented our approach in Python and ran our experiments single-threaded on a laptop with a 2 GHz Intel Core i5 processor and 16GB of memory.

Configurations

Our approach requires a buffer size b that can store all UI-events belonging to a single task. We report on the results using a buffer size of 250 UI-events (also for the streaming baselines), which covers three times the number of events of the longest task in our data collection. Furthermore, we consider that parts of our approach are initialized at the beginning of a stream and populated over time: the global co-occurrence matrix and control-flow-counts for task identification and the online clustering model for task categorization. Given that the accuracy of these components may improve as more UI-events are observed, we test the value of a *warm-up phase*, where our approach populates the matrix, control-flow counts, and clustering model using the first 0, 100, 250, 500, or 1,000 UI-events, before starting to identify and categorize on them. Note that, in offline settings, the number of warm-up events corresponds to the number of UI-events in the input log.

Baselines

We compare against three streaming baselines and two offline baselines with respect to task identification and categorization. Note that we cannot compare against any baseline with respect to object-instance identification, because none of the existing works considers that as a part of recognizing tasks.

Streaming baselines. We compare our approach to three baselines in a streaming setting. Aside from our initial approach [151], we established two baselines by adapting existing works, since there are currently no other techniques capable of recognizing tasks based on a stream of user interactions. The task-identification component of the two additional baselines consists of an existing, offline identification technique, lifted to an online setting (as described below). Their task-categorization components, by contrast, are operationalized with the same technique used in our approach. This is necessary because existing offline categorization techniques cannot be lifted to an online setting.

- BL_{caise}: *Our previous approach.* In our previous work [151], we proposed the first approach for recognizing task-level events from user interaction streams, which we use to investigate the value of our extended task-identification component.
- BL_{dfg}: *Back-edge-based identification.* Leno et al. [108] proposed a log segmentation technique based on back-edges identified from a directly-follows graph (DFG). We adapted the technique to build the DFG incrementally using the same UI-classes as available to our approach and apply the authors' back-edge detection method periodically, after every b events (i.e., each time the buffer is full).
- $BL_{co\text{-}oc}$: *Co-occurrence-based identification.* The approach by Urabe et al. [177], which also inspired parts of our work, leverages co-occurring UI-classes in fixed windows to segment a log. We adapted it to count co-occurrence incrementally and compute similarities on a buffer of UI-events. We use the same parameter configurations as reported in their paper.

Offline baselines. When evaluating our approach in offline settings, we compare it against two baselines:

- BL_{leno}: *Back-edge-based identification.* The original approach by Leno et al. [108] serves as the first offline baseline. As described for BL_{dfg}, it constructs a DFG based on UI-classes that immediately follow each other in a given log. Then, it detects back-edges in the graph, which it uses to segment the log into tasks. As such, this baseline only covers task identification.
- BL_{urabe}: *Co-occurrence-based identification and subsequent clustering.* The original approach by Urabe et al. [177] consists of two phases: task identification and task clustering. First, it segments the complete input log using co-occurring UI-classes (as explained above), before applying agglomerative hierarchical clustering to categorize tasks. As such, it covers both task identification and task categorization.

Measures

We use the following measures to assess quality in our experiments.

Object-instance-identification quality. We assess object-instance-identification quality through the well-known precision, recall, and F_1-measures by comparing the object instances our approach identified from a stream/log to the object instances in the manually created gold standard. Using A to denote the set of object instances identified by our approach and G for the set of object instances in the gold standard these measures are defined as follows:

- *Precision* (*Pre.*). Precision is the fraction of object instances that are actually correct ($|A \cap G| / |A|$).

- *Recall* (*Rec.*). Recall is the fraction of object instances in the gold standard that were also correctly identified by our approach ($|A \cap G| / |G|$).
- F_1*-score* (F_1). The F_1-score is the harmonic mean of precision and recall.

Task-identification quality. We assess task-identification quality by comparing the identified task segments to those of the corresponding gold standard, for which we use two measures:

- *#tasks.* To assess if an approach makes the right amount of segmentation decisions, we compare the numbers of identified and gold-standard tasks.
- *Normalized edit distance* (*n.ED*). To quantify how similar identified tasks are in comparison to the gold standard, we calculate the average normalized edit distance between identified tasks and their closest task in the gold standard.

Task-categorization quality. We assess categorization quality through measures for cluster quality, by comparing the tasks that are assigned to the same category in the gold standard (i.e., task types).

Cluster quality.

- *Rand index* (*R*). We compute the Rand index, which considers the fraction of pairs (tasks at macro level, UI-events at micro level) that are correctly assigned to the same or to different categories, i.e., $(TP+TN) / (TP+TN+FP+FN)$, where a true positive (TP) indicates that two tasks/events are correctly assigned to the same category.
- *Jaccard index* (*J*). We also compute the weighted average Jaccard index to quantify the similarity between identified clusters and the gold-standard clusters, which is given as $A \cap G / A \cup G$ per cluster, with A a cluster's contents identified by our approach and G its gold-standard contents (i.e., tasks at the macro level, UI-events at the micro level).

5.3.3 Results

In this section, we first present the overall results of our approach applied in an online setting in comparison to the streaming baselines, before examining the impact of a warm-up phase. Following this, we discuss the importance of considering various types of information in task identification through an ablation study. Shifting our focus to offline settings, we report on our approach's performance in comparison to the offline baselines, followed by a discussion on performance differences between online and offline settings. Lastly, we discuss computational efficiency.

Online Results

In Table 5.3, we present the online results of our approach in terms of object-instance identification, while Table 5.4 shows the outcomes of task-identification and task-categorization. The latter table includes results for the three streaming

baselines and a perfect identification strategy (to show the quality of task categorization independently of task-identification quality), with a warm-up phase of 250 UI-events. Note that the baselines do not take object-instance identification into account, making it unfeasible to derive corresponding results for this aspect.

Table 5.3: Results of our approach for object-instance identification.

Stream	Approach	Pre.↑	Rec.↑	F_1 ↑
S_{U1}	*Ours*	0.91	0.88	0.90
S_{U2}	*Ours*	0.95	0.93	0.94
S_{U3}	*Ours*	-	-	-

Table 5.4: Results of our approach and the baselines for task-identification and categorization (warm-up of 250 UI-events). *Perf. ident.* shows task-categorization results in case of perfect task identification. ↑ and ↓ indicate the desired direction per measure.

		Task Ident.		Task Categorization			
Stream	**Approach**	**#tasks**	**n.ED↓**	**R(mi)↑**	**R(ma)↑**	**J(mi)↑**	**J(ma)↑**
S_{U1}	*Ours*	**202**	**0.04**	**0.96**	**0.97**	**0.92**	**0.95**
	BL_{caise}	**198**	**0.04**	0.94	0.95	**0.92**	0.92
	BL_{dfg}	**202**	0.83	0.58	0.82	0.38	0.89
	$BL_{co\text{-}oc}$	159	0.33	0.74	0.80	0.64	0.71
	Perf. ident.	200	0.00	1.00	1.00	1.00	1.00
S_{U2}	*Ours*	**241**	**0.05**	**0.96**	**0.97**	**0.92**	**0.95**
	BL_{caise}	231	0.06	0.89	0.95	0.82	0.89
	BL_{dfg}	99	0.32	0.42	0.80	0.24	0.56
	$BL_{co\text{-}oc}$	198	0.33	0.69	0.71	0.50	0.51
	Perf. ident.	240	0.00	0.97	0.97	0.95	0.95
S_{U3}	*Ours*	**132**	**0.17**	**0.97**	**0.99**	**0.93**	**0.99**
	BL_{caise}	138	0.23	0.87	0.88	0.76	0.76
	BL_{dfg}	29	0.58	0.34	0.49	0.16	0.35
	$BL_{co\text{-}oc}$	58	0.37	0.45	0.57	0.31	0.46
	Perf. ident.	120	0.00	1.00	1.00	1.00	1.00

Object-instance-identification results. Our approach achieves accurate results in terms of object-instance identification for S_{U1} and S_{U2}, both with an F_1-score of 0.9 or higher. The similar precision and recall scores (0.91 resp. 0.88 for S_{U1} and 0.95 and 0.93 for S_{U2}) further suggest that the object-instance-identification component is both accurate in identifying actual object instances and effective in

identifying most of these object instances. Note that for S_{U3}, we could not obtain any results, because there is simply no object instance information available to be identified. Furthermore, recall that none of the baselines considers object-instance identification, thus, we could not obtain respective results for those either.

Looking at the results in detail reveals that our approach is capable of correctly identifying objects of various types. For instance, it correctly extracts objects of the type *bank account*, identified by an account number, and of type *tax code*, identified by a respective code. However, there are also cases where our approach fails, in particular because object types were not recognized correctly. Type recognition relies on the accurate identification of nouns; hence, failing to correctly identify nouns hinders the recognition of object instances. For instance, noun identification failed to recognize that *address* is a noun, as it can also function as a verb, which leads to relevant instances being missed. Similarly, it does not recognize types that are not represented by actual words, e.g., *countycode* (which misses white space between *county* and *code*). Conversely, although discarding UI-specific objects (such as *button* or *text field*) generally avoids false positives, our approach identified some instances that do not have gold standard counterparts. For instance, it identified *personal data* objects, associating these with an identifier. However, such objects are not informative and their identification should, thus, be avoided.

Overall, the results still show that our approach can accurately identifying object instances from user interaction streams, allowing it to relate tasks to each other.

Task-identification results. Our approach achieves highly accurate results for S_{U1} and S_{U2}, identifying approximately the same numbers of tasks as in the gold standard (202 vs 200 and 241 vs 240), to which they are very close in terms of contents, yielding edit distances of just 0.04 and 0.05. However, stream S_{U3} is more challenging. Our approach overestimates the total number of tasks (132 versus 120), achieving an edit distance of 0.17. An in-depth look into the results shows that it occasionally fails to recognize that certain sub-tasks belong to the same gold-standard task, since they lack contextual relatedness and overlapping data values.

Compared to the baselines, our approach consistently obtains better results in terms of edit distances. This indicates that the tasks that the baselines identify differ more from their gold-standard counterparts than the ones identified by our approach. Our previous approach, BL_{caise}, performs similarly well as our approach for S_{U1} and S_{U2} (with 0.007 improvement for S_{U1} and 0.01 improvement for S_{U2}), yet considerably better for S_{U3} (improved by 0.06). This improvement can be attributed to the additional check for deterministic control-flow that our approach employs, which BL_{caise} does not do. BL_{dfg} and $BL_{co\text{-}oc}$ often miss segmentation points, resulting in much lower numbers of identified tasks than contained in the gold standard. BL_{dfg}, specifically, only finds 99 tasks for S_{U2} (out of 240) and 29

for S_{U3} (out of 120). Although $BL_{co\text{-}oc}$ generally performs better than BL_{dfg}, we find that its results are heavily dependent on the selection of two parameter values, with the edit distances differing by up to 0.5 across configurations.[11]

Overall, these results indicate that our approach, which considers the semantic and data perspectives in addition to the control-flow perspective considered by BL_{dfg} and $BL_{co\text{-}oc}$, leads to more accurate task identification. Furthermore, our approach does not depend on user-defined parameters (unlike $BL_{co\text{-}oc}$).

Task-categorization results. As for task categorization, our approach achieves high macro Rand scores of 0.97 for S_{U1} and S_{U2} and 0.99 for S_{U3}, which shows that it accurately assigns pairs of tasks to the same category as their gold-standard counterparts. The comparable micro-level scores show that this categorization quality generally also holds for pairs of UI-events, which thus accounts for tasks of different lengths. The Jaccard index, which provides insights into the quality per cluster, rather than per task (or UI-event) pair, confirms the accurate categorization quality, achieving macro scores of 0.95 for S_{U1} and S_{U2} and 1.00 for S_{U3} and the comparable scores on the micro level (0.92, 0.92, and 0.97).

As shown by the results obtained when using our task-categorization component on perfectly identified tasks (gray row in Table 5.4), the categorization itself is highly accurate, achieving perfect scores for S_{U1} and S_{U3}, and near-perfect ones (≥ 0.95) for S_{U2}. The improved task-categorization performance on S_{U3} compared to BL_{caise} can, therefore, be attributed to our improved task-identification component. The subpar results of BL_{dfg} and $BL_{co\text{-}oc}$, which use the same categorization technique as our approach, also clearly indicate that lower identification quality leads to lesser categorization results.

Impact of the Warm-up Phase

Table 5.5 shows the results of our approach for warm-up phases of 0, 100, 250, 500, and 1,000 UI-events. The results indicate that the warm-up phase does not have any impact on task-identification quality. For task-categorization quality, the benefit of a warm-up phase becomes clear, though.[12] A closer look at the streams suggests that this benefit relates to whether the warm-up phase covers each task type at least once. For S_{U1}, this coverage occurs within the first 250 UI-events but not within the first 100 UI-events. Between these two warm-up phases we see a clear performance boost. While for up to 100 warm-up events, we achieve a macro Rand score of 0.84 and Jaccard score of 0.78 for S_{U1}, setting the warm-up phase to 250 UI-events increases the scores by 0.13 resp. 0.17. A further extension of the warm-up to 500 still improves the results, yet, less substantially (by

[11] See our repository for detailed experiments regarding $BL_{co\text{-}oc}$'s parameter configurations.

[12] Warm-up phases do not apply for object-instance identification as this is done per UI-event.

Table 5.5: Results of our approach with warm-up phases of 0, 100, 250, 500, and 1,000 UI-events. ↑ and ↓ indicate the desired direction per measure.

		Task Ident.		Task Categorization			
Stream	**#Events**	**#tasks**	**n.ED↓**	**R(mi)↑**	**R(ma)↑**	**J(mi)↑**	**J(ma)↑**
S_{U1}	*0*	202	0.04	0.89	0.84	0.84	0.78
	100	‘’	‘’	0.89	0.84	0.84	0.78
	250	‘’	‘’	0.96	0.97	0.92	0.95
	500	‘’	‘’	**0.98**	**0.99**	**0.97**	**0.98**
	1,000	‘’	‘’	0.98	0.99	0.97	0.98
S_{U2}	*0*	241	0.05	0.96	0.97	0.92	0.95
	100	‘’	‘’	0.96	0.97	0.92	0.95
	250	‘’	‘’	0.96	0.97	0.92	0.95
	500	‘’	‘’	0.96	0.97	0.92	0.95
	1,000	‘’	‘’	**0.98**	**0.99**	**0.97**	**0.99**
S_{U3}	*0*	132	0.17	0.90	0.93	0.81	0.87
	100	‘’	‘’	0.91	0.94	0.84	0.88
	250	‘’	‘’	**0.97**	**0.99**	**0.93**	**0.99**
	500	‘’	‘’	0.97	0.99	0.93	0.99
	1,000	‘’	‘’	0.97	0.99	0.93	0.99

0.02–0.03). For S_{U3}, where all task types are covered by the first 100 UI-events, performance improves as well. However, this improvement is not as considerable as observed for S_{U1} (macro scores increase by only 0.01). Notably, increasing the warm-up phase to 250 UI-events has a more significant impact on categorization quality for S_{U3} (macro scores increase by 0.05–0.11). This may be attributed to the coverage of additional execution variants per task type, of which S_{U3} has significantly more compared to other streams (30 versus at most 7). Interestingly, for S_{U2}, performance remains consistently high regardless of whether all task types are seen during warm-up or not. This suggests that it is not strictly necessary to have observed all task types before starting categorization to achieve good quality.

Overall, these results show that a warm-up phase is not necessary for task-identification, but that it can be beneficial for task categorization, if an application context allows for it. However, even without any warm-up phase our approach achieves good categorization results across streams.

Ablation Study

Our task-identification component uses various types of information associated with UI-events to decide whether a task completes or continues (cf. Section 5.2.3).

In order to understand the value of taking information beyond the control-flow into account, we conducted an ablation study. This involves, in turn, removing those task-identification strategies that consider data values (including object instances), semantic information (specifically overhead actions), and both of these.

Table 5.6: Task-identification results of the ablation study. ↑ and ↓ indicate the desired direction per measure.

Stream	S_{U1}		S_{U2}		S_{U3}	
Considered perspectives	**#tasks**	**n.ED↓**	**#tasks**	**n.ED↓**	**#tasks**	**n.ED↓**
Full approach	**202**	**0.04**	**241**	**0.05**	**132**	**0.17**
Control-flow & semantic	**202**	**0.04**	**241**	**0.05**	150	0.27
Control-flow & data	302	0.35	338	0.33	**132**	**0.17**
Control-flow only	336	0.42	372	0.38	150	0.27

We present the results of the ablation study in Table 5.6. It shows that we achieve the same identification performance on S_{U1} and S_{U2} when removing the data values from consideration, whereas the performance on L_{U3} decreases (the edit distance worsens from 0.17 to 0.27). Conversely, when not considering the semantic perspective, we obtain considerably worse results S_{U1} and S_{U2} (the edit distance worsens by ca. 0.3 for both streams), while achieving the same performance as when considering all perspectives on S_{U3}. Finally, the subpar results achieved when only considering control-flow information highlight the value of taking additional perspectives into account.

Overall, the results indicate that considering specific perspectives is important for some streams and not for others. However, only when considering all perspectives can the overall good performance of our approach across streams be achieved.

Offline Results

Table 5.7 shows the results of our approach applied in an offline setting compared to BL_{leno} and BL_{urabe}. For BL_{urabe}, the table shows the average results across the configurations that were evaluated in the original paper [177] as well as the results of the best configuration per log.

The results show that our approach also outperforms the offline baselines by a large margin. When it comes to task identification, it achieves an edit distance of just 0.04 for L_{U1}, while the baselines fall short; BL_{urabe} achieves only 0.28 at best and BL_{leno} achieves 0.35. For L_{U2}, BL_{urabe} achieves an edit distance of 0.28 in the best case and BL_{leno} achieves 0.26, while our approach achieves 0.05; a substantial improvement of 0.23 compared to BL_{urabe} and 0.21 compared to BL_{leno}.

Table 5.7: Results of our approach applied in an offline setting and the offline baselines, BL_{leno} and BL_{urabe}. Note that BL_{leno} only provides task identification and that, for BL_{urabe}, we show average results across the configurations used in the original paper [177] and the results from the best configuration per log. *Perf. ident.* shows task-categorization results in case of perfect task identification. ↑ and ↓ indicate the desired direction per measure.

		Task Ident.		**Task Categorization**			
Log	**Approach**	**#tasks**	**n.ED↓**	**R(mi)↑**	**R(ma)↑**	**J(mi)↑**	**J(ma)↑**
L_{U1}	*Ours*	**202**	**0.04**	**0.98**	**0.99**	**0.97**	**0.99**
	BL_{leno}	50	0.35	-	-	-	-
	BL_{urabe} (avg.)	163	0.56	0.72	0.78	0.59	0.64
	BL_{urabe} (best)	131	0.28	0.90	0.98	0.85	0.97
	Perf. ident.	200	0.00	1.00	1.00	1.00	1.00
L_{U2}	*Ours*	**241**	**0.05**	**0.98**	**0.99**	**0.97**	**0.99**
	BL_{leno}	101	0.26	-	-	-	-
	BL_{urabe} (avg.)	190	0.54	0.69	0.72	0.48	0.53
	BL_{urabe} (best)	161	0.28	0.89	**0.99**	0.80	**0.99**
	Perf. ident.	240	0.00	1.00	1.00	1.00	1.00
L_{U3}	*Our approach*	**132**	**0.17**	**0.97**	**1.00**	**0.93**	**1.00**
	BL_{leno}	90	0.48	-	-	-	-
	BL_{urabe} (avg.)	25	0.67	0.26	0.36	0.11	0.28
	BL_{urabe} (best)	18	0.66	0.26	0.54	0.11	0.52
	Perf. ident.	120	0.00	1.00	1.00	1.00	1.00

The performance gains are even more considerable for L_{U3}, where we observe an improvement of 0.49 compared to BL_{urabe} and 0.31 compared to BL_{leno}. This again highlights the efficacy of our task-identification strategy that considers control-flow, semantic, and data information, instead of solely relying on control-flow.

Interestingly, BL_{urabe} achieves good macro-level results for task categorization on L_{U1} and L_{U2} (0.97–0.99) considering the subpar task-identification results. This indicates that the baseline's post-hoc categorization generally works well, despite poor task-identification quality. For L_{U3}, for which the tasks are more similar in terms of their UI-classes across types, the baseline's poor identification quality cannot be compensated by its good categorization performance, yielding macro Rand and Jaccard scores of 0.52 at best. In contrast, our approach achieves high macro (0.99–1.00) and micro (0.93–0.99) scores for all three logs. Note that BL_{leno} does not cover task categorization, for which we could, thus, not compute results.

Overall, both good task-identification and task-categorization quality are required to accurately abstract a user-interaction log to a task-level event log, which our approach provides across the evaluation logs.

Performance in Online Versus Offline Settings

As shown in the previous sections, our approach achieves consistently high results in both online and offline settings. Specifically, its object-instance and task-identification performance is equal in both settings. Only for task categorization with short warm-up phases, performance decreases slightly when applied in online compared to offline settings. For the approach by Leno et al. [108], we observe a clear performance drop when comparing the results of the adapted online version (BL_{dfg}) to the original approach (BL_{leno}), showing that it is more suitable for offline task identification. However, we do not observe the same trend for the approach by Urabe et al. [177], which also targets offline task identification and categorization. Although its offline version (BL_{urabe}) generally outperforms the adapted online version ($BL_{co\text{-}oc}$) in terms of task categorization, it is noteworthy that its online versus, on average, achieves better results for task identification.

Overall, the evaluation results highlight that, regardless of the specific setting, our approach achieves good performance in recognizing task-level events from user interaction data. It consistently outperforms the baselines in both online and offline settings and, unlike existing works, allows to relate recognized events to each other.

Computational Efficiency

Finally, we assessed the memory and response time efficiency of our approach. To assess memory efficiency, we measure the maximum memory it requires, which is the sum of the largest buffer size during run time, the final size of the global co-occurrence matrix, the directly-follows counts between UI-class sets, and the final size of the clustering model. As for response time, we measure how long it takes our approach to perform object-instance identification and task identification after an UI-event arrives, as well as how long it takes to categorize an identified task.

We find that our approach requires less than 1% of the memory that would be needed to store all UI-events from the streams, thus clearly demonstrating its memory efficiency. As for response time, our approach requires between 1 and 107 ms, for object-instance identification, between 2 and 4 ms for task identification, and between 40 to 150 ms for task categorization. Note that the latter is only executed once per identified task. Therefore, the response time depends on a task's length, i.e., number of UI-events it consists of. Given that the average time between user interactions is over 2.5 seconds in the available data, this means that our approach can easily keep up in terms of responses.

5.4 Limitations

Our experiments have shown that our approach is capable of transforming low-level user interaction data into task-level events that can be used for process mining. Nevertheless, our work is subject to certain limitations, which relate to the approach itself and its evaluation.

As for our approach, a key limitation is that it assumes that tasks are executed sequentially, i.e., one task must be completed before another one is started. This is naturally a limiting assumption as it is well-imaginable that users switch between tasks in their daily work. Especially knowledge workers in office settings commonly work on several tasks in an interleaving manner [29]. However, it is important to remark that this limitation so far applies to all unsupervised approaches that recognize tasks from user interaction data, because it is highly challenging and may impose additional data requirements to be solved properly.

As for our evaluation, we acknowledge that the considered user interaction data may impact generalizability of the results. Although this data covers a variety of task types, was obtained from different sources, and goes beyond the evaluation data used in other work [177], it does not capture a user's real sequence of process-related and overhead tasks conducted during a workday. Therefore, we plan to conduct further experiments as soon as more suitable data becomes available.

Furthermore, the generalizability of the sets of completion actions, keywords for overhead actions, and UI-objects (all used during task identification) remains to some degree uncertain given the available data. While these sets are generic, stem from established design guidelines, and occur across different types of graphical user interfaces, we cannot guarantee their completeness for any user interaction stream or log. Nevertheless, they can easily be adjusted and extended so that they, for instance, cover domain-specific applications and different languages.

5.5 Related Work

Our work primarily relates to research on the identification of tasks from low-level event data (Section 5.5.1), RPM (Section 5.5.2), the identification of object-centric information from event data (Section 5.5.3), and preprocessing techniques for stream-based process mining (Section 5.5.4).

5.5.1 Identifying Tasks from Low-level Event Data

Various approaches have targeted the identification of tasks in low-level event data. Focusing on user interaction events, various works [19, 127] take a supervised approach based on the computation of alignments between user interaction logs and

task models that they require as input. Tello et al. also approach task identification in a supervised manner by applying classical machine learning approaches [174], whereas Pegoraro et al. train a neural network model to segment user interaction logs [137, 138]. Linn et al. [118] combine transactional data recorded by information systems with user interaction logs, to integrate interaction data with traditional process mining. Finally, our earlier work [155] identifies tasks through self-learning of multi-perspective dependencies between low-level interactions, although this currently requires expensive pre-training on large interaction logs.

Beyond user interaction events, related approaches aim to recognize process-related tasks from so-called active-window-tracking data [29], ambient or wearable sensors [46, 152, 154], network traffic data [76, 91], or low-level, server-side application logs [79]. However, due to the low-level, abstract nature of the data used by these approaches, they depend on supervised recognition strategies or even manual labeling, as opposed to our unsupervised approach.

5.5.2 Robotic Process Mining

The core idea of RPM is to discover repetitive routines from user interaction logs, which are suitable for automation [109]. Leno et al. [109] propose an RPM-pipeline, which starts from a raw user interaction log and eventually yields automatable task scripts. The so-called *segmentation stage* of this pipeline identifies which user interaction events jointly form individual tasks (yet, not their types), thus only partially solving part of the transformation problem that our approach addresses. We use the corresponding approach by Leno et al. [108] as a baseline for task identification in our evaluation. The *candidate-routine-identification stage* (also covered by Leno et al. [108]) recognizes tasks that are executed in the same or similar manner. In a sense, tasks are thus categorized into candidate routines, which may be considered as types. However, this stage only takes tasks into account if they are performed frequently, thus neglecting infrequent ones. Furthermore, it does not assign labels to identified tasks. Unlike candidate-routine identification, the approach by Urabe et al. [177] categorizes all tasks that it previously identified through segmentation, thus addressing task identification and categorization. However, the approach does not relate tasks to each other in any way. We lift the segmentation strategy of this approach to an online setting and use it as a baseline in our evaluation.

Although our work and works on RPM (partially) address similar sub-problems, their overarching goals are fundamentally different. RPM aims to get in-depth insights into the execution of individual tasks with the goal of automating suitable ones, whereas we aim for a comprehensive transformation of low-level user interactions into task-level events that are usable in process mining settings.

5.5.3 Identifying Object-centric Information in Event Data

Relating task-level events to each other by identifying to which case they belong has been addressed by several works [60]. The problem of inferring missing object information from event data as well as the conversion of classical event logs into object-centric logs have also been investigated [156, 170]. Berti et al. propose approaches to extract object-centric event logs from SAP systems and relational databases [31]. Finally, the extraction of object-centric information from knowledge graphs has been researched [183]. However, these techniques assume that events are already on the task level and that they can operate in an offline setting, whereas our approach overcomes both of these assumptions.

5.5.4 Stream-Based Preprocessing of Event Data

Research on stream-based preprocessing in process mining mostly focuses on cleaning noisy event streams. Van Zelst et al. [186] filter a stream based on estimates of how likely new events belong to real process behavior, whereas Hassani et al. [94] filter noise by extracting frequent sequential patterns from an event stream before applying streaming process discovery. Finally, Awad et al. [25] propose an approach to resolve situations in which events arrive in an incorrect order on a stream. However, these techniques assume arriving events to be on the task level, even though, in practice, streaming data is commonly at a lower-level of abstraction, such as taken into account by our approach.

5.6 Summary

In this chapter, we proposed an automated approach for recognizing task-level events from user interaction data, which works in a fully unsupervised manner. It segments user interaction data to identify tasks, categorizes these according to their type, and relates tasks to each other via object instances it extracts from the low-level events. In this manner, our approach creates task-level events that meet the requirements of process mining settings as they relate to a process-level activity and to a specific process execution. In addition to being the first approach that addresses this transformation problem in full, it works in both, offline and streaming settings. In particular, it takes a user interaction stream and recognizes task-level events that it emits to a task-level event stream. Streaming process mining techniques can then subscribe the task-level stream and process its events on the fly.

We demonstrated our approach's efficacy in an experimental evaluation on real user interaction data and showed that it outperforms three streaming baselines and two offline baselines in both their scope and accuracy.

Chapter 6

Object-Information Extraction from Event Logs

Many real-world processes comprise multiple concurrent object types with complex interrelations [6]. For example, in an order-handling process, multiple items can be part of a single order and multiple orders can be shipped in one package. As such, individual steps in the process may involve any number of different objects, such as sending items of multiple orders in one package. Therefore, such steps cannot be assigned to a single process execution and thus, their corresponding events cannot be assigned to a single case either. This violates the basic assumption of case-centric event logs (cf. Section 2.1.1). If event data of such processes is nevertheless captured in a case-centric format, this can cause data quality issues, such as duplicate events and spurious behavioral relations, which ultimately lead to inaccurate analysis results.

However, researchers and practitioners have long exclusively focused on case-centric event logs when developing and applying process mining techniques. Consequently, there is an abundance of event data captured in such logs, without access to the source systems from which these were extracted (cf. [67, 69]). As a result, the only option to resolve the data quality issues that are caused by the log format, is to transform a case-centric log into an object-centric one. Performing this transformation manually is a tedious and time-consuming task, considering the complexity of real-world logs, with dozens of attributes and thousands of events. Hence, the transformation needs to be supported automatically.

In this chapter, we propose an approach that addresses this problem by automatically transforming a case-centric event log into an object-centric one. Such a transformation is far from straightforward, though, because it requires an approach to identify which object types occur in the event log, which object instances exist

with which properties, and how these instances relate to the events. This information is to a certain extend contained in case-centric event logs, but in an unstructured, i.e., hidden way. Our approach uncovers this information by combining the semantic annotation of events (using the approach we proposed in Chapter 3) with data profiling and control-flow-based relation-extraction techniques.

This chapter is based on a paper titled "*Uncovering Object-centric Data in Classical Event Logs for the Automated Transformation from XES to OCEL*" [156] by Adrian Rebmann, Jana-Rebecca Rehse, and Han van der Aa.

The remainder of the chapter is structured as follows. Section 6.1 illustrates the challenges that our approach needs to address. Section 6.2 presents our approach itself. Section 6.3 describes our evaluation, which shows that our approach can accurately rediscover object-centric logs that were transformed into case-centric ones and can effectively mitigate quality issues in real-world logs. Section 6.4 reflects on limitations of our work. Section 6.5 discusses related work and Section 6.6 provides a summary.

6.1 Problem Illustration

This section illustrates the problems caused by recording object-centric event data in case-centric event logs and the challenges that must be overcome when transforming such logs into object-centric counterparts. For this, we use the example of an order-handling process [6], which involves four types of objects: *customers*, *orders*, *items*, and *packages*. As shown in Figure 6.1, a customer can place multiple orders, an item belongs to exactly one order and one package, a package can contain multiple orders, and an order can be split over multiple packages.

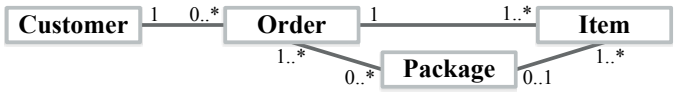


Figure 6.1: UML data model of the running example.

Problems of case-centric event logs. We illustrate the problems of recording a multi-object process in a case-centric format using the following trace, with an order as the case notion, i.e., the main object type from whose point of view the process execution is recorded.

$\sigma_{order} = \langle$*Create order*, *Reorder item*, *Pick item*, *Send package*, *Pick item*, *Send package*$\rangle$

The events in σ_{order} indicate the picking of two items and the sending of two packages. Although their ordering suggests that these activities occur in an interleaving

fashion, there is a clear relation between first picking an item and then sending it in a package. This clear precedence relation on the item level is lost, because there can be several items and packages per order, which we cannot distinguish on the trace level. This phenomenon, called *divergence*, is unavoidable when recording processes with object relations beyond 1:1 in the form of case-centric event logs [6]. It often occurs together with another unavoidable issue, called *convergence*. Convergence emerges when we use an individual item as the case notion to represent the events from trace σ_{order}, which results in the following traces:

$\sigma_{item}(1) = \langle$*Create order*, *Reorder item*, *Pick item*, *Send package*$\rangle$
$\sigma_{item}(2) = \langle$*Create order*, *Pick item*, *Send package*$\rangle$

Because both items belong to the same order, the *Create order* event is duplicated across the traces. As a result, the information that both items relate to the same order is no longer captured at the trace level. Due to the m:n relations in the process at hand, the impact of this issue is amplified, given that also multiple orders can relate to the same package, and vice versa.

To overcome these issues and the incorrect analysis results they lead to, a case-centric log needs to be transformed into an object-centric one, as discussed next.

From case-centric to object-centric logs. To illustrate the transformation of case-centric into object-centric event logs, consider the example in Table 6.1, which provides a case-centric event log with two orders. The log captures information on the events related to each order, as well as attributes that associate events with a `PackageID`, a `Weight`, and the `Customer`.

Table 6.1: Two traces of a case-centric event log of an order handling process with the *order* as the case notion.

CaseID	Event	Activity	Timestamp	PackageID	Weight	Customer
o1	e1	Create order	05-20 09:07			Pete
o1	e2	Reorder item	05-23 10:40		12.5	Pete
o1	e3	Pick item	05-23 14:20		70.8	Pete
o1	e4	Send package	05-23 17:26	p1	70.8	Pete
o1	e6	Pick item	06-04 15:20		12.5	Pete
o1	e9	Send package	06-06 16:20	p2	20.4	Pete
o2	e5	Create order	06-03 19:17			Pete
o2	e7	Update order	06-04 18:11			Pete
o2	e8	Pick item	06-05 11:48		7.9	Pete
o2	e10	Send package	06-06 16:20	p2	20.4	Pete

Table 6.2: Object-centric event log of the running example.

Event	Activity	Timestamp	Orders	Packages	Items	Customer
e1	Create order	05-20 09:07	{*o1*}	∅	{*i1_1*,*i1_2*}	{*Pete*}
e2	Reorder item	05-23 10:40	{*o1*}	∅	{*i1_1*}	{*Pete*}
e3	Pick item	05-23 14:20	{*o1*}	∅	{*i1_2*}	{*Pete*}
e4	Send package	05-23 17:26	{*o1*}	{*p1*}	{*i1_2*}	{*Pete*}
e5	Create order	06-03 19:17	{*o2*}	∅	{*i2_1*}	{*Pete*}
e6	Pick item	06-04 15:20	{*o1*}	∅	{*i1_1*}	{*Pete*}
e7	Update order	06-04 18:11	{*o2*}	∅	{*i2_1*}	{*Pete*}
e8	Pick item	06-05 11:48	{*o2*}	∅	{*i2_1*}	{*Pete*}
e9	Send package	06-06 16:20	{*o1*,*o2*}	{*p2*}	{*i1_1*,*i2_1*}	{*Pete*}

Table 6.3: Objects of the object-centric event log.

Type	Instances
Customer	{*Pete* ()}
Order	{*o1* (), *o2* ()}
Package	{*p1* (`Weight`: 70.8), *p2* (`Weight`: 20.4)}
Item	{*i1_1* (`Weight`: 12.5), *i1_2* (`Weight`: 70.8), *i2_1* (`Weight`: 7.9)}

As shown in Table 6.2 and Table 6.3, constructing an object-centric version of this event log requires information about: object types (customers, orders, items, and packages), their instances and associated properties (e.g., that package *p1* has a weight of 70.8), and the relations between object instances and events (e.g., that event *e1* creates order *o1*, which relates to items *i1_1* and *i1_2*). However, such crucial information is not explicit in the case-centric version of the event log, but rather needs to be uncovered in order to transform case-centric data into an object-centric log. This results in four main transformation tasks:

1. **Detect object types.** Object types in a process are not explicitly indicated in case-centric event logs. Rather, transformation requires these types to be extracted from unstructured activity labels, such as the *order* type in "*Create order*", and from certain event attributes, such as `Customer` in Table 6.1.
2. **Identify object instances.** Due to divergence and convergence, a transformation approach needs to identify distinct object instances within cases, e.g., that case *o1* deals with two items and two packages, and relate object instances across cases, e.g., that package *p2* appears in both *o1* and *o2*. This involves identifying event attributes that represent identifiers of a specific object, e.g., that `PackageID` defines individual packages. Furthermore, because such identifier attributes may not exist for all object types, it also requires in-

ferring certain object instances from the event log itself, e.g., that events *e3* and *e6* yield two different items (*i1_1* and *i1_2*).

3. **Relate objects to their properties.** Case-centric event logs do not distinguish between attributes that relate to a specific event, such as a resource performing it, and attributes that provide information about the object handled in the event, such as the `Weight` attribute, which captures information about an individual package or item. When establishing an object-centric log, such relations must thus be derived by separating event attributes from object properties, in order to have a comprehensive view on the instances involved in the process, as captured in Table 6.3.
4. **Associate object instances with events.** Finally, instead of referring to a specific case, each event in an object-centric log must be mapped to the object instances it relates to. Obtaining a complete mapping requires a thorough analysis of the inter-relations that exist between object instances. For example, this requires the recognition that package *p2* relates to orders *o1* and *o2*, as well as items *i1_1* and *i2_1*, and associating all these objects with event *e9*, even though the objects originally stem from a range of different events and cases in the case-centric log.

6.2 Extraction Approach

As shown in Figure 6.2, our approach for the transformation of a case-centric event log L (cf. Definition 3) into an object-centric event log OL (cf. Definition 5) consists of five main steps.

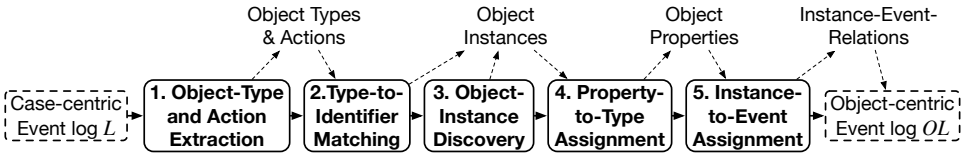


Figure 6.2: Overview of the transformation approach.

Step 1 extracts the *object types and actions* from the activity label and other textual attributes of events, which yields object types and the applied actions per event. Steps 2 and 3 jointly establish a set of object *instances*: Step 2 first matches extracted object types to attributes that capture identifiers to recognize distinct instances of object types, whereas Step 3 aims to discover instances for object types for which no such identifier attribute was found. Afterwards, Step 4 aims to assign *properties* to object types by identifying attributes that represent object properties.

Finally, Step 5 assigns the discovered *object instances to events* by exploiting behavioral relations among object types and instances discovered in previous steps. Based on the result of Step 5, we create an object-centric event log according to the *OCEL* format [83][1]. Next, we describe each of these five steps in detail.

6.2.1 Object-Type and Action Extraction

The first step of our approach extracts the object types and actions from an event log. Object types need to be derived from unstructured textual attribute values, such as activity labels, and attribute names of a case-centric event log. An *action* is applied to an object, incurring a change in its state [110]. For instance, the *Create order* activity indicates that a *create* action is applied to an *order*. We extract actions along with the object types since these can contain information about the creation of new object instances, which we will exploit in a later step.

To achieve this, we use our semantic-annotation approach introduced in Section 3.3. Recall that it extracts parts of textual attribute values that correspond to different semantic roles, such as *object types* and *actions* in two ways:

1. *Instance-level labeling* labels parts of unstructured textual attribute values with semantic roles. The parts that correspond to the desired roles are then extracted. For instance, for the *Send package* activity label of event *e9*, it labels *package* as an object type and *send* as an action.
2. *Attribute-level classification* identifies event attributes that in their entirety correspond to a certain semantic role. It does so based on an attribute's name and its value range. This, e.g., applies to the `Customer` attribute in Table 6.1, which allows us to also identify *customer* as an object type contained in the event log, assigning this type to any event that has a value for the attribute.

By taking the output of the semantic annotation approach, Step 1 instantiates a function `extract`, which, given an event $e \in E_L$, extracts the object types and the actions applied to them (if any) from e. The result maps the object types to a (possibly empty) set of actions, e.g., $\texttt{extract}(\textit{e9}) = \{\textit{package} \rightarrow \{\textit{send}\}, \textit{customer} \rightarrow \emptyset\}$. Each event's object map is then initialized with its types, e.g., $\textit{e9.objects} = \{\textit{package} \rightarrow \emptyset, \textit{customer} \rightarrow \emptyset\}$. The empty set in the object maps are later used to store references of a respective event to object instances. Finally, we establish a set of identified object types $OT = \bigcup_{e \in E_L} \texttt{dom}(e.\textit{objects})$ and move to Step 2, which aims to match these types to identifier attributes.

[1]Note that our approach is not limited to this specific object-centric event-log format and can be adapted to output logs in alternative formats, such as *DOCEL* as we showed in our other work [85].

6.2.2 Type-to-Identifier Matching

In this step, our approach tries to associate identifier attributes with the extracted object types to be able to recognize distinct object instances. For our running example, we can differentiate between the two packages *p1* and *p2* by recognizing that the `PackageID` is an identifier for *package* objects. Such identifier attributes are not explicitly given, meaning that we need to match object types to attributes. To establish these matches, our approach first identifies a set $\mathcal{D}_L^{ID}$ of potential object identifiers, by categorizing attributes according to their domain. Then, we match these attributes to object types in OT, resulting in a set Φ, consisting of (ot, d) pairs with $ot \in OT$ and $d \in \mathcal{D}_L^{ID}$.

Finding potential identifier attributes. To identify the set $\mathcal{D}_L^{ID} \subseteq \mathcal{D}_L$, we recognize that identifiers generally use alphanumeric domains, i.e., `string` or `int`, such as `PackageID` (*p1* and *p2*) and `Customer` (*Pete*) in Table 6.1. Therefore, we categorize attributes according to their domain's data type and add those with `string` and `int` domains to $\mathcal{D}_L^{ID}$. This way, we discard attributes corresponding to, e.g., timestamps, Boolean values, and floats, such as the `Weight` attribute.

Matching identifier attributes and object types. Next, we identify matches between the object types in OT and potential identifier attributes in $\mathcal{D}_L^{ID}$, resulting in a set of matches Φ. Some object types and attributes can be directly matched. For others, we first establish candidate matches, and then verify their validity.

Direct matching. Object types in OT that stem from the attribute-level classification of Step 1 reflect objects that correspond to the name of a specific event attribute, such as the *customer* object type corresponding to the `Customer` attribute. Because these types were identified in this manner, we know that their identifiers are captured in the corresponding attributes, if these are part of $\mathcal{D}_L^{ID}$. Therefore, we directly add such pairs, e.g., (*customer*, `Customer`), to the matches in Φ.

Establishing candidate matches. For object types that cannot be directly matched, such as the *package* type extracted from activities, we first establish candidate matches, collected in a set $\Phi_{candidate}$, considering attribute names and values.

First, we establish candidate matches by checking if the name of an unmatched object type encompasses the name of an unmatched attribute, or vice versa. E.g., we recognize a candidate match between the *package* type and the `PackageID` attribute, or between the *item* type and a, hypothetical, `order_item` attribute.

Then, for attributes in $\mathcal{D}_L^{ID}$ that are not yet in a candidate match with an object type, we apply a *data profiling*-strategy [14], which checks if an attribute exclusively *co-occurs* with an object type. For the running example, all events associated with the *package* type (*e4*, *e9*, *e10*) have values for the `PackageID` attribute, but this attribute does not apply to any other events. Thus, even if `PackageID` was

named `pID` and hence not a name-based candidate match, our approach would still recognize it as a potential identifier for the *package* type and add it to $\Phi_{candidate}$.

Validating candidate matches. Although name-based similarity and co-occurrence are useful indicators to identify relations between object types and attributes, there is no guarantee that the candidate matches capture proper identifiers. Therefore, we next validate each candidate match $(ot, d) \in \Phi_{candidate}$ by determining if each unique value of d is associated with a specific instance of ot, and vice versa.

This validation task is complex, though, given that multiple events in a case can relate to the same object instance (e.g., creating and updating an order) or multiple instances of the same object type (e.g., shipping multiple packages for one order), and that, due to duplication issues, the same event can essentially appear in two cases (cf. *e9* and *e10*). For an object type ot, we deal with these issues by aiming to establish a set of events $E'_L(ot)$ that should each relate to a different instance of ot. Given $E_L(ot) \subseteq E_L$ as the events related to ot (i.e., that have ot in their *objects* attribute after Step 1), we obtain $E'_L(ot)$ by avoiding duplicate events and by selecting only a single event per case. Our approach avoids duplicates by only selecting events from $E_L(ot)$ that have a unique combination of an activity label, timestamp, and event attributes (aside from their `CaseID` and event ID, if available). In this manner, we detect *e9* and *e10* as duplicates. Given the identified duplicates, we select a single event per case related to ot, in a manner that maximizes the size of $E'_L(ot)$. For instance, given $E_L(package) = \{e4, e9, e10\}$, we select *e10*, because *e4* and *e9* stem from the same case, and obtain $E'_L(package) = \{e4, e10\}$.

Finally, if the attribute values of d are unique for the events in $E'_L(ot)$, we consider d as a valid identifier of ot and add (ot, d) to Φ. For instance, we consider (*package*, `PackageID`) a valid match, given that the two events in $E'(package)$ have unique values for the attribute, *p1* and *p2*. If there are multiple valid candidates for the same object type, we match the type to the attribute with the largest number of unique values and discard the other candidates.

Object-instance creation. For all matches $(ot, d) \in \Phi$, our approach creates an object instance o with its type ot for each unique value of d, i.e., its identifier oi, and adds these instances to the object maps of the events that refer to this instance. For example, we add package *p1* to event *e4* and package *p2* to events *e9* and *e10*.

6.2.3 Object-Instance Discovery

Next, our approach sets out to discover instances for those object types for which no explicit identifier attribute was found in the previous step, such as the *item* type in the running example. For this, we try to find activities that indicate the instantiation of objects, either based on their activity labels or based on the life-cycle of an object.

Instance discovery based on creation actions. First, the approach identifies activities whose meaning hints at the instantiation of an object, such as *Create order*. To this end, we again use the action classification framework of the MIT Process Handbook [122]. It defines a set of *creation actions* (see Table 6.4) describing the creation of some output.

Table 6.4: Creation actions [122] used by object-instance discovery.

build	*compute*	*construct*	*copy*	*create*
design	*develop*	*document*	*duplicate*	*generate*
make	*manufacture*	*perform*	*produce*	*record*

Given an event, we check if any of its actions, extracted in the first step, corresponds to an action in this set. If so, the occurrence of this event implies the instantiation of a new object. For instance, we recognize that events *e1* and *e5*, corresponding to the *Create order* activity label, result in two new orders.

Although we here identify creation actions based on the MIT Process Handbook, our work is independent of this specific resource. It can be replaced or enhanced with alternatives, such as the *build verbs* from the framework by Levin [115], multilingual resources, e.g., *ConceptNet* [168], or a self-defined set.

Instance discovery based on object life-cycles. Although creation actions are a reliable indicator for the creation of new objects, they are not always available for an object type. Therefore, our approach next analyzes the life-cycles of object types in terms of the applied activities per case, which is illustrated in Figure 6.3.

life cycle 1 ⟨Receive request, Update request, Complete request⟩
life cycle 2 ⟨Receive request, Receive request, Complete request, Complete request⟩

Indicator activity: *"Receive request"*

Figure 6.3: Recognizing activities that indicate new object instances.

For an object type ot, we aim to identify an *indicator activity*, which corresponds to a new object instance. We look for such an indicator by checking if there are any activities related to ot that occur for every case of this type. For example, assuming only the two depicted cases relate to requests in the process at hand, both *Receive request* and *Complete request* are candidate activities, since they occur in both life-cycles. In case of such a tie, we select the activity that most commonly occurs first among the candidates—*Receive request* in the example—as the indicator activity that we use to identify new object instances. Therefore, we recognize that the cases in Figure 6.3 relate to three distinct *request* objects: one in the first life-cycle and two in the second.

Object-instance creation. For each event that indicates a new object instance, based on a creation action or indicator activity, our approach establishes an object instance, for which we generate a unique identifier oi, and add it to the event's object map. Duplicate events, as identified in Step 2, form an exception here. Since they correspond to the creation of the same object instance, which is why we assign the same instance to them. Note that we discard all object types for which neither Step 2 nor Step 3 identified any instances, by removing the type from OT as well as from any event's object map.

6.2.4 Property-to-Type Assignment

In this step, our approach tries to associate properties to object types, which are attributes that capture information about an object instance associated with an event, rather than relate to the event itself. For instance, although event *e3* (*Pick item*) has a `Weight` attribute with a value of 70.8, it is clear that this refers to the weight of the item, not of the event. Therefore, in this step we aim to establish a mapping between a log's attributes $\mathcal{D}_L$ and the object types in OT.

To establish this mapping, we first select all attributes that were not recognized as object identifiers in Step 2. Then, we consider an attribute d to be a property of an object type ot if (1) events related to ot have a value for d and (2) all events related to the same object instance have the same value for d. The former ensures co-occurrence, ascertaining that d indeed relates to ot, whereas the latter ensures that object properties are immutable per object instance, in line with their definition in the OCEL format [83]. In this manner, we identify that `Weight` is an attribute of both the *item* and *package* types, whereas attributes such as a `timestamp` or `employee` are not identified as properties, because they change across the events related to the same object instance.

Finally, we avoid assigning an attribute d as a property to multiple object types if the attribute name indicates a clear relation to one of the types. For example, we avoid assigning an `item_category` attribute to the *package* type, given that this property clearly relates to items, irrespective of the co-occurrence of the attribute and packages.

6.2.5 Instance-to-Event Assignment

Finally, our approach sets out to complete the mapping between events and object instances, which is necessary to account for missing *instance-to-event* and *instance-to-instance* relations. The former involves events that correspond to a particular object type, but for which no particular instance has yet been discovered. For example, event *e2* (*Reorder item*) is already recognized as relating to the *item*

type, yet we still need to identify that this event refers to the same item that is later handled by event *e6* (*Pick item*). The latter refers to the inter-relations that can exist among object instances, which need to be reflected in the object maps of the corresponding events. For example, since package *p1* relates to order *o1*, event *e4*, which creates this package, should also be associated with that order. We identify these missing relations as follows.

Finding missing instance-to-event relations. To find missing instance-to-event relations, we first identify the events that are associated with an object type (through Step 1), but for which no instance was discovered in Step 2 or 3. This applies, e.g., to event *e2* (*Reorder item*) and *e7* (*Update order*). Then, given such an event, we search within the case for other events that are associated with an object instance of the same type and verify that the object's properties match across the events. For instance, since event *e2* has a `Weight` of 12.5, we do not want to associate it with the item of event *e3*, which has a weight of 70.8, but rather with the same item as event *e6*, which also relates to an item weighing 12.5kg. Should multiple object instances satisfy this requirement, we associate the event to the instance of its nearest predecessor or successor.

Finding missing instance-to-instance relations. Our approach finds missing instance-to-instance relations by (1) considering relations between instances and cases, (2) identifying strict orders among object types, and (3) consolidating cross-case relations.

Discovering case objects. We first exploit that, commonly, each case in a case-centric event log corresponds to an instance of a particular object type, such as an *order* in our running example. If such a *case object* can be identified, we know that any other object instance handled in the same case also relates to that object, e.g., that the items and packages handled in the first case all relate to order *o1* as well.

However, to recognize such inter-relations, we need to identify the case object type, if any, for a particular log. Given that instances of this object type must, by definition, be in a 1:1 relation with the cases in a log, we first discard all object types for which this does not apply, i.e., which are affected by convergence and divergence. Given an object type ot, we thus ensure that (1) no instance of ot is associated with multiple cases, such as the *package* type in the example, and (2) that no case is associated with multiple instances of ot, such as the *item* type.

If these checks yield a single case object type, ot_c, we add each object instance of ot_c to the object map of all events e in their case, using the case's id as the instance's identifier. For example, *order* is the only object type that passes the checks, so that we assign orders *o1* and *o2* to all events in their respective cases.

Strict order between object types. We next identify instance-to-instance relations by looking for the existence of strict orders between object types. Here, we consider an object type ot_1 to be in a strict order with type ot_2 if every time an event

related to ot_2 occurs, an event related to ot_1 (directly or indirectly) precedes it. In this manner, we, for example, observe a strict order between items and packages in the running example. Note that these object types can still occur in an interspersed manner, as seen in case *o1*, where events related to items occur between packages.

Given a strict order between ot_1 and ot_2, we relate an instance o_1 of type ot_1 to an instance o_2 of ot_2 if the life-cycle of o_1 completes before the one of o_2 begins, i.e., if the last event related to o_1 comes before the first event related to o_2. For example, we relate *i1_1*, which last occurs in *e6*, to *p2*, which first occurs in *e9*.

Consolidating cross-case relations. Last, we consolidate inter-relations across cases by ensuring that duplicate events are associated with the same sets of object instances. Given two duplicates, e and e', we achieve this by associating both events with all object instances stemming from the union of their object maps. For example, having recognized that events *e9* and *e10* are duplicates, we add all object instances stemming from case *o1* (associated with *e9*) to the object map of *e10*, and vice versa. In this manner, we, e.g., recognize that package *p2*, which is created by these duplicate events, deals with item *i1_1* (stemming from case *o1*) as well as item *i2_1* (stemming from *o2*).

Having associated events with all object instances, our approach has uncovered the necessary information to construct its output, an object-centric event log OL.

6.2.6 Output

Our approach returns an object-centric event log according to the OCEL format [83], which, at a high level, consists of an *objects* and an *events* map[2].

The *objects* map relates object identifiers to instances, which are in turn associated with their type and property values. To populate this map, we add all instance identifiers, either detected in Step 2 or generated in Step 3, to the map and associate these instances with their properties identified in Step 4. Simultaneously, we also disassociate any object property from the events that they were associated with in the case-centric event log, e.g., rather than having `Weight` as an attribute of event *e4*, we represent it as a property of the respective package: $\mathit{objects}[\mathit{p1}] = \{\mathit{package}, \{\texttt{Weight:}70.8\}\}$.

The *events* map associates identifiers with events, which are associated with object instances through their *objects* attribute. It is important to recognize that these events are no longer grouped per case. As a result, we can omit any duplicate event from consideration, e.g., by removing event *e10* and preserving *e9*. The map is then populated with the remaining events, which are each associated with the

[2]Note that our conceptual approach is not bound to specific output format and can be extended to consider, for instance, dynamic object properties as well [85].

identifiers of their respective object instances, as assigned in Steps 2, 3, and 5, e.g., $e1.objects = \{order = \{o1\}, item = \{i1\_1, i1\_2\}$.

Based on the established maps, we return the object-centric log, which can directly be used by object-centric process mining techniques [10, 117].

6.3 Evaluation

We implemented our approach in Python[3], using the *PM4Py* library [32] for event log handling. Based on this implementation, we perform experiments to assess our approach's capability to rediscover object-centric logs that were transformed into case-centric ones. Finally, we illustrate its practical value by showing that it can resolve divergence and convergence in real-world scenarios.

6.3.1 Data Collection

For our evaluation experiments, we use a publicly available OCEL log of an order handling process.[4] It contains 22,367 events and 11,522 object instances of five object types: 2,000 orders, 8,159 items, 20 products, 17 customers, and 1,325 packages. From this original OCEL log, we create three case-centric logs, using the *item*, *order*, or *package* as the case notion. The resulting logs capture 1:n, n:1, and n:m relationships between objects and include object types both in attribute names and activity labels. Thus, all relation types are covered, meaning that all strategies employed by our approach can be assessed.

6.3.2 Evaluation Setup

To assess the ability of our approach to correctly discover relevant object-centric information in case-centric event logs, we conduct experiments using two settings:
(1) All attributes. In this setting, we use all information from the case-centric event logs as input for the rediscovery task.
(2) Masked ID attributes. To assess the robustness of our approach, we also purposefully reduce the information that is available by masking object ID attributes in the case-centric event logs. This increases the dependency of our approach on its instance discovery techniques employed in Step 3. Specifically, we mask each ID attribute once for each of the three case-centric logs. Since the *item* and *order* logs include identifiers for all four other types, and the *package* log captures only a

[3] `https://github.com/a-rebmann/object-information-extraction`
[4] `http://ocel-standard.org/1.0/running-example.jsonocel.zip`

customer identifier, we obtain nine *masked* logs, one with *package*, four with *item*, and four with *order* as the case notion.

We measure the performance of our approach in terms of the well-known *precision*, *recall*, and F_1*-score* metrics with respect to the original OCEL log per type of element, i.e., object types, object instances, properties, and instance-to-event assignments. Using A to denote the set of elements uncovered by our approach and G for the set of elements in the ground truth, i.e., the OCEL log, precision is the fraction of elements uncovered by our approach that are actually correct ($|A \cap G|/|A|$), recall is the fraction of elements in the OCEL log that were also correctly uncovered by our approach ($|A \cap G|/|G|$), and the F_1-score is the harmonic mean of precision and recall. Because transforming an object-centric into a case-centric log can cause the loss of information about entire object types, we only include object types in G that are actually contained in a particular case-centric log. To avoid propagating false positives from object-type extraction (Step 1), we only include elements in A that relate to object types actually present in the original OCEL log for the other steps.

6.3.3 Evaluation Results

Table 6.5 reports on the results of our rediscovery experiments with the *all attributes* setting, whereas Table 6.6 reports on the results with the *masked ID attribute* setting; all numbers are micro-averaged over the logs. In the following, we discuss the results for the different tasks that our approach addresses.

Table 6.5: Results of the experiments (*all attributes*) averaged over logs.

Element	Count	Precision	Recall	F_1
Object Types	12	0.71	1.00	0.82
Object Instances	24k	1.00	1.00	1.00
Object Properties	10	0.37	1.00	0.54
Instance-to-Event	411k	0.94	1.00	0.97

Table 6.6: Results of the experiments (*masked ID attributes*) averaged over logs.

Element	Count	Precision	Recall	F_1
Object Types	42	0.71	1.00	0.82
Object Instances	94k	1.00	1.00	1.00
Object Properties	34	0.39	1.00	0.56
Instance-to-Event	1,559k	0.93	0.97	0.95

Object-type extraction. For the extraction of object types, our approach achieves a recall of 1.00 and a precision of 0.71, yielding an F_1-score of 0.82. We thus accurately identify all object types from the original log. The lower precision is caused by the extraction of two additional object types, *payment reminder* and *delivery*. Although not contained in the original OCEL log, their extraction is not problematic and can even enable additional insights, e.g., on the number of payment reminders sent per order.

Object-instance identification. Our approach identifies object instances with perfect accuracy in both the regular and masked settings. This highlights its ability to find and match ID attributes to object types (Step 2) and the usefulness of our instance-discovery strategies (Step 3), which can identify instances for types with masked ID attributes.

Property-to-type assignment. When assigning properties to object types, our approach achieves a perfect recall, but a rather low precision of 0.37. An in-depth look reveals that these different assignments are not problematic, though. For example, the attribute `cost` is assigned to both *product* and *item*, whereas in the original it is only associated with products. However, given that also items have costs, such assignments are redundant, but not wrong. Similarly, our approach associates attributes such as `price` and `weight` with orders, items, packages, and products. While these are realistic assignments, the attributes are not considered as properties in the original OCEL log, but are associated with events. Thus, our approach actually provides a more complete mapping.

Instance-to-event assignment. For instance-to-event assignment, we achieve an excellent recall (0.998, rounded in Table 6.5) and a high precision (0.94) in the all-attributes setting. Thus, our approach assigns relevant object instances to events they relate to. An in-depth look into the constructed OCEL logs reveals that the superfluous assignments of instances to events are mainly assignments of packages to events that relate to items shipped in the respective package. Such assignments are not considered in the original log, but can enable insights into the packaging process in a post-hoc analysis.

When masking identifier attributes (Table 6.6), precision and recall decrease slightly, which indicates that our approach occasionally makes incorrect assignments. This is especially the case for 1:n relationships between object types and the case notion. For example, in the *order* event log, where one order may contain many different items, items with the same properties may be assigned incorrectly. However, it is important to recognize that such assignments are simply not possible based on the information in the masked log, whether done by an automated approach or manually.

Our evaluation experiments show that our approach is capable of accurately uncovering object-centric information from artificially created case-centric event

logs, using different settings and case notions. We also observe that our approach even uncovers more information than originally captured in the OCEL log. This includes additional object types, properties, and relations, which allow for deeper insights into the process. The main difficulty for our approach was the recognition of object inter-relations for objects in a 1:n relation with the case object, which resulted in several incorrect instance-to-event assignments. Despite their promise, the evaluation results must be considered with care, given that only one original OCEL log was available as a basis.

6.3.4 Application Cases

To demonstrate the practical value of our approach, we show that it can resolve convergence and divergence in well-known real-world event logs. The full results and OCEL logs obtained by our approach can be found in our repository (see page 137). In the following, we use individual cases and events from these logs to illustrate in detail how our approach mitigates divergence and convergence.

Divergence. We use the BPI17 application log [67] to show how our approach mitigates divergence issues. The log captures a loan application process, containing 1,202,267 events, 31,509 cases, and 26 distinct activities. Divergence is particularly frequent, because the log uses the *application* as the case notion and one application can have multiple offers. This means that cases in the log often contain multiple events that denote execution of the same activity for distinct offers (divergence). Applying process discovery to the log leads to loop-backs, as visualized in Figure 6.4. This shows the DFG discovered for one case of the log, which is already quite complex.

When applying our approach to mitigate the divergence issue, we discover that 42,995 offers are handled in the 31,509 applications and that offers have several properties, such as an offered amount and a monthly cost. For the particular case in Figure 6.4, we find that four distinct offers are handled in this application, that these all have different properties, and that the process is linear with respect to a single offer, e.g., $\langle$*Create Offer*, *O Created*, *O Sent*, *O Canceled*$\rangle$. It is important to stress that this information on the sub-case level is not readily available in the case-centric log and has to be uncovered by identifying the distinct offers handled in a single case. Our approach achieves this by extracting the *offer* type, finding an identifier attribute for it, and assigning it, among others, the `MonthlyCost` and `OfferedAmount` properties.

Convergence. To illustrate how our approach can mitigate convergence issues, we chose the BPI19 event log [69], which captures data on the purchasing process of a multinational company and contains 1,595,923 events across 251,734 cases

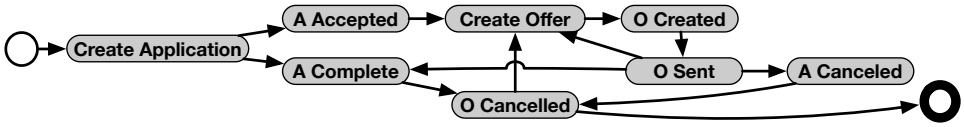


Figure 6.4: DFG of application 196483749 of the BPI17 log.

with 42 distinct activities. Each event relates to a single *purchase order item* and multiple purchase order items can belong to the same *purchasing document*. Consequently, events on the purchasing-document level are duplicated across cases (convergence). For example, the duplication of "*Vendor creates invoice*" events suggests the creation of invoices per purchase order item, whereas in reality invoices can cover multiple such items.

When applying our approach to mitigate convergence, we discover, among others, 251,734 purchase order items, 76,349 purchasing documents, 86,868 invoices, 1,975 vendors, and 4 companies. The resulting OCEL log reveals the relationships between object types, as shown in Figure 6.5: Purchasing documents consist of any number of purchase order items, a vendor creates multiple invoices, and each invoice is associated with one purchasing document. Notably, in contrast to the input log, events related to purchasing documents and invoices, such as *Vendor creates invoice* or *Document created*, are not captured at the level of individual purchase order items, but at the level of purchasing documents, thus eliminating duplicate events. This demonstrates that our approach can reveal actual relationships among objects and mitigate the convergence issue present in real-world logs.

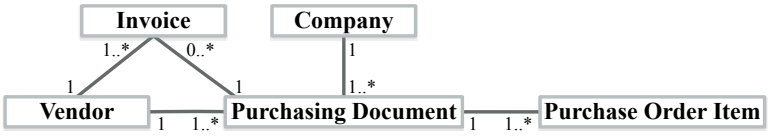


Figure 6.5: UML model with object-type relations found in the BPI19 log.

These application cases demonstrate that our approach can mitigate divergence and convergence in real-world event logs. Although, due to a lack of a ground truth, the completeness of uncovered object-centric information cannot be quantified, the results nevertheless show that our approach provides considerable practical value by extending the analysis potential for event data of multi-object processes.

6.4 Limitations

Our experiments reveal that our approach is capable of recovering object-centric information from case-centric event logs and can effectively mitigate quality issues related to case-centric event data. In its current form, our work has certain limitations, though, that relate to the approach and the evaluation.

First, object types must at least be mentioned in the case-centric log for our approach to extract them. However, once an object type is extracted, instances, properties, and relations can be identified through the use of diverse strategies that include and go beyond the semantic analysis of events. Second, to accurately handle n:1 and m:n relations with respect to the case notion, our approach relies on duplicate detection, which requires (non-duplicate) events to have discriminative timestamps or attribute values. Finally, because the assignment of objects to events often depends on domain knowledge about inter-object relations, our approach can currently not handle all scenarios. For example, it is not clear without domain knowledge that items should relate to packages but not vice versa.

Despite their promise, the quantitative evaluation results must be considered with care, given that only one suitable object-centric log was available as a basis. Moreover, while we obtain the case-centric event logs for this evaluation by transforming an object-centric log, in reality such logs are typically established through a dedicated extraction procedure and preprocessing (cf. Section 1.1). Due to a lack of such data with a corresponding gold standard, we cannot assess our work in such scenarios in a quantitative manner. As for the qualitative evaluation, the lack of a gold standard and limited domain knowledge does not allow us to claim completeness of uncovered object-centric information either.

6.5 Related Work

Our work primarily relates to research on object-centric representations of event data and discovering object-centric information from event logs.

6.5.1 Object-Centric Representations of Event Data

After storing event data in case-centric formats like XES [88] for many years, the first data format proposed for object-centric event logs was the *eXtensible Object-centric Event Log* (XOC) format [116]. It does not require a case notion and therefore avoids forcing multi-dimensional data into cases. More recently, researchers introduced the OCEL format [83], which allows for more efficient storage and processing than its predecessor. Beyond log formats, another proposed option for storing multi-dimensional object-centric event data are event graphs, which enable

the analysis of behavior of different objects handled in a process [77]. For our approach, we adopt OCEL as the output format, which, among others, enables the subsequent application of techniques for discovering object-centric process models, such as object-centric behavioral constraint models [117] and object-centric Petri nets [10].

6.5.2 Discovering Object-Centric Information from Event Logs

Approaches for the discovery of object types and their behavioral relations from event data usually require relational data or rich logs that cover multiple perspectives of a process as input. This includes approaches for the discovery of artifact (i.e., object) life-cycles from raw logs of artifact-centric systems [74, 145] as well as the discovery of behavioral dependencies between object types based on such logs [144] or based on data extracted from ERP systems [120]. Compared to these approaches, our approach takes case-centric event logs, where no explicit relations between objects are given, and transforms them into object-centric logs. The approach by Bano et al. [27] uses case-centric event logs as input data as well. However, their goal is to discover UML models from activity labels and attribute names to provide analysts with domain-specific context information not to transform event data in order to enable object-centric process analysis.

6.6 Summary

In this chapter, we proposed an approach to uncover object-centric data from case-centric event logs to automatically transform them into object-centric logs. To this end, our approach combines the semantic analysis of textual event attributes with data profiling and control-flow-based relation-extraction techniques. It extracts object types, discovers object instances and their properties, and assigns these instances to events they relate to.

We demonstrated our approach's efficacy in an evaluation by showing that it is able to rediscover an object-centric log from case-centric event logs that were generated from it. Furthermore, we showed that it can mitigate the well-known convergence and divergence issues in real-world event logs. In this manner, the approach can alleviate the problem of obtaining incorrect results as a consequence of capturing event data in an unsuitable log format. Furthermore, it enables object-centric process mining based on existing case-centric event logs.

Chapter 7

Best-Practice-Violation Detection in Event Logs

Detecting undesired process behavior in event logs can reveal data quality issues, compliance problems, and process inefficiencies. In process mining settings, this is primarily done using conformance-checking techniques [41], which require dedicated normative models that capture what desired behavior entails in a given process. Here, the normative process model is compared with the traces of an event log in order to reveal any deviations that might have occurred. While such techniques can provide valuable insights for organizations, an inherent problem is that such dedicated process models are rarely available to them and require time-consuming and costly efforts in their creation [72]. Fortunately, many process types, such as procurement and invoicing processes, are commonly organized in similar ways across organizations or (at least) involve similar steps. As a consequence, *reference process models* have been recognized as an important means to provide depictions of proven ways to run these processes, serving as best-practice templates for process implementations in various domains [80, 86]. Furthermore, they contain knowledge about general behavioral relations that have to hold in these processes, e.g., that an invoice must be checked before it is approved. The availability of reference process models can, thus, alleviate the need for manually creating a customized normative model, as they provide a basis to check for undesired behavior.

However, finding a single reference model for a real-world event log is impractical because the individual needs of organizations can vary. This means that a real-world process may be subject to additional requirements than those captured in a reference model and that some parts of a reference model may not be applicable in a particular situation. Additionally, event logs may cover the behavior of multiple reference models, e.g., related to both procurement and invoicing. There-

fore, in order to leverage the best practices captured in these models for detecting undesired behavior, a more flexible alignment between them and an event log must be established, involving multiple reference models at once.

In this chapter, we tackle this problem by proposing an approach that mines declarative constraints from a reference model collection, automatically selects constraints for a given event log, and then checks for constraint violations. The constraints we extract may stem from thousands of reference process models that capture best practices of a plethora of domains, depending on the model collection used as input. Here, the main challenge is that we need to determine which constraints are actually relevant and interesting—or even applicable—for a given event log. We address this challenge through established techniques from declarative process mining as well as NLP methods to refine and measure the relevance of mined constraints for a given event log. We implement our approach and demonstrate its capability to detect best-practice violations through an evaluation based on real-world process model collections.

This chapter is based on a manuscript titled "*Mining Constraints from Reference Process Models for Detecting Best-Practice Violations in Event Logs*" [153] by Adrian Rebmann, Timotheus Kampik, Carl Corea, and Han van der Aa, which is under review at the time of writing.

The remainder of the chapter is structured as follows. Section 7.1 illustrates the potential and problem of mining constraints from reference process models to check for best-practice violations. Section 7.2 presents our approach for mining and selecting relevant constraints and checking for violations using these. We evaluate in Section 7.3 in a quantitative manner and apply it in application cases using real-world event logs. Section 7.4 reflects on limitations of our work. Finally, Section 7.5 discusses related work and Section 7.6 provides a summary.

7.1 Problem Illustration

This section illustrates the potential and problem of detecting best-practice violations using declarative constraints that are mined from reference process models.

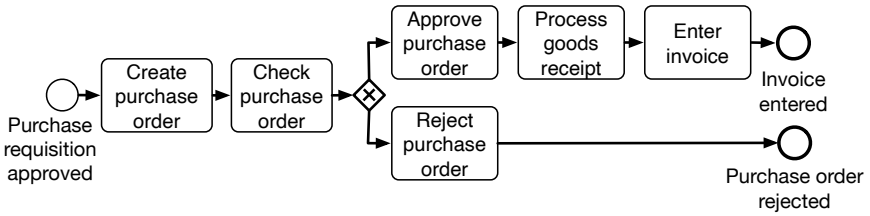


Figure 7.1: Exemplary reference process model in BPMN.

Detecting best-practice violations using a reference process model. Consider the process model in Figure 7.1, which shows an exemplary reference process model of a procurement process for materials. From this model, we can extract a variety of behavioral relations, such as: A *purchase order (PO)* must be *created* before it is *checked* and, also, *checked* before it is *approved*; a *PO* must not be both *approved* and *rejected*; a *PO* must be *approved* before *goods* are *received*; and no *invoice* should be *entered* when a *PO* is *rejected*.

Now, consider a trace σ that stems from an event log of a procure-to-pay process for consumer products:

$\sigma = \langle$*create order, receive goods, enter invoice, check amount, make payment*$\rangle$

In this trace, an *order* is first *created*, before *goods* are *received*, then an *invoice* is *created* of which the *amount* is *checked* and, finally, a *payment* is *made*. We can use the relations extracted from the reference process model to identify behaviors in σ that correspond to best-practice violations. For instance, *goods* are *received* in σ but an *order* was neither *checked* nor *approved* beforehand and an invoice is *entered* before *goods* were *received*. While the latter may depend on the specific organizational context, e.g., an arrangement with a supplier would allow for entering an invoice before goods are delivered, the former is clearly undesirable, regardless of any specific arrangements.

The example thus shows that the reference model can be used to gain insights on best-practice violations of the given trace, despite not stemming from exactly the same process. In fact, some of the behavioral relations captured in reference models can be broadly applicable. For example, the fact something cannot be both *approved* and *rejected* applies to virtually any process in which such approval decisions need to be made.

Combining constraints from multiple reference process models. An important aspect to recognize, though, is that, besides containing activities that relate to materials procurement, trace σ also contains activities that go beyond the scope of the reference process model shown in Figure 7.1. In particular, σ also contains activities related to *checking the invoice amount* and *making a payment*, which would sooner be covered by a reference model relating to an invoicing process. Therefore, relying solely on a single reference model may result in missing relevant best-practice violations, which means that it is necessary to jointly check for adherence to behavior captured in multiple reference models.

This can be achieved by turning the behavioral relations captured by imperative process models (such as Figure 7.1) into declarative constraints. This has two main benefits: (1) it allows to consider behavioral relations from multiple reference models at the same time and (2) it allows to omit parts of the imperative process model, that are not relevant to the trace at hand, from consideration. Specifically, in

the presence of tens of thousands of constraints, we need to avoid that constraints that are irrelevant for an event log cause the detection of violations, i.e., *false positives*. For instance, a constraint such as "*process goods receipt* should precede *enter invoice*" (extracted from a procurement process model) should not be applied to a sales process event log, which does not involve *goods receipt* objects.

However, expecting users to manually select from tens of thousands of constraints is unfeasible. To reduce manual effort, we need to automatically preselect constraints based on the specific context, i.e., the given event log. To this end, our approach mines constraints from a large collection of reference process models and only selects those constraints that are relevant to the particular situation at hand.

7.2 Detection Approach

This section presents our proposed approach for detecting best-practice violations in event logs based on constraints mined from reference process models. As illustrated in Figure 7.2, it consists of three main stages.

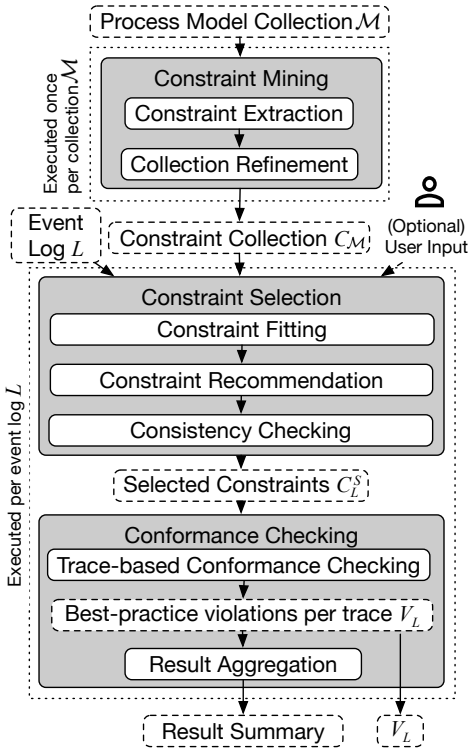


Figure 7.2: Overview of the detection approach.

The *Constraint Mining* stage takes as input a collection of reference process models $\mathcal{M}$ (according to Definition 7), from which it extracts a set $C_{\mathcal{M}}$ of best-practice constraints. This first stage is independent of a specific event log and, therefore, only needs to be performed once for a given collection $\mathcal{M}$. Then, given an event log L, the *Constraint Selection* stage identifies which constraints from the collection $C_{\mathcal{M}}$ are applicable to log L and fits these constraints to the log, resulting in a set of selected, fitted constraints C_L^S. This stage contains multiple sub-stages, including constraint identification and checking whether the recommended set of constraints is consistent. Note that this stage can be executed in a fully automated manner, although users can also choose to manually refine the set C_L^S. Finally, the *Conformance Checking* stage compares the traces in log L against the constraints in C_L^S. This results in a map of detected best-practice violations V_L that assigns each trace σ in L a set of best-practices that it violates. Furthermore, it provides a summary of violating traces per constraint type. In the following, we describe the individual stages of our approach in detail.

7.2.1 Constraint Mining

In the constraint mining stage, we establish a constraint collection $C_{\mathcal{M}}$ that represents best practices found in a reference process model collection $\mathcal{M}$. In the remainder of this section, we first describe the constraint types and templates that our approach covers, before discussing the steps performed to mine the constraints: *Constraint Extraction* and *Collection Refinement*.

Coverage

Our approach covers four types of constraints, corresponding to constraints that capture behavioral relations that should hold between activities (i.e., activity-level constraints), across object types (i.e., inter-object constraints), within object types (i.e., intra-object constraints), and ones that restrict the organizational roles that can perform particular process steps (i.e., role constraints). Examples of each constraint type are shown in Table 7.1. To represent the specific constraints, we use different sets of DECLARE templates per constraint type, as shown in Table 7.2.

Activity-level constraints. Activity-level constraints capture best practices about the behavioral relations that should hold between pairs of activities in a trace. In order to represent different activity inter-relations, we use DECLARE templates that capture that two activities should occur together (RESPONDEDEXISTENCE, CO-EXISTENCE) or in a certain order (SUCCESSION, ALTERNATESUCCESSION), an activity causes (RESPONSE, ALTERNATERESPONSE) or requires (PRECEDENCE,

Table 7.1: Constraint types covered by our approach with examples.

Constraint type	Exemplary constraint	Description
Activity-level	PRECEDENCE(*Approve PO*, *Process GR*)	*Approve PO* should happen before *Process GR*.
	NOTCOEXISTENCE(*Reject PO*, *Process GR*)	*Reject PO* and *Process GR* should not co-occur.
	SUCCESSION(*Process GR*, *Enter Invoice*)	*Process GR* happens if (and only if) *Enter Invoice* happens later.
Inter-object	PRECEDENCE(*purchase order*, *goods receipt*)	A *purchase order* should appear before a *goods receipt*.
	RESPONDEDEXISTENCE(*invoice*, *goods receipt*)	If there is an *invoice* there should be a *goods receipt* as well.
Intra-object	PRECEDENCE(*create*, *approve*) | *purchase order*	For each *purchase order*, *create* should precede *approve*.
	ATLEASTONE(*create*) | *purchase order*	For a *purchase order*, *create* should always occur.
Role	ABSENCE(Enter invoice) if `role` $\neq$ *accounts payable*	*Enter Invoice* should be performed by *accounts payable*.

ALTERNATEPRECEDENCE) another activity, or two activities should not occur together (NOTCOEXISTENCE).

For instance, an activity-level constraint may capture that an *approve purchase order* activity must precede a *process goods receipt* activity, which is captured by a PRECEDENCE(*approve purchase order*, *process goods receipt*) constraint. We do not cover constraints that require activities to immediately occur after each other (e.g., CHAINRESPONSE) which, we assume, is too restrictive for best practices. Each such activity-level constraint is defined as follows:

Definition 14 (Activity-level constraint) *An activity-level constraint $c_{\mathcal{M}} \in C_{\mathcal{M}}$ is a tuple $c_{\mathcal{M}} = (type, templ, a_1, a_2, support)$, with $c_{\mathcal{M}}.type = activity$, $c_{\mathcal{M}}.templ$ the constraint's* DECLARE *template, $c_{\mathcal{M}}.a_1$ and $c_{\mathcal{M}}.a_2$ the activities, and $c_{\mathcal{M}}.support$ the constraint's support.*

Note that the constraints of all types we cover have a $support$ that indicates how often a constraint was extracted from the model collection $\mathcal{M}$ and is set when refining the collection of mined constraints $C_{\mathcal{M}}$.

Inter-object constraints. Inter-object constraints capture relations that should hold between pairs of object types occurring in a process, e.g., that a trace can

Table 7.2: Templates used per constraint type (cf. Table 2.4 for their definition).

Constraint types	**Templates**
Activity, Inter-object	RESPONDEDEXISTENCE, PRECEDENCE, ALTERNATEPRECEDENCE, RESPONSE, ALTERNATERESPONSE, SUCCESSION, ALTERNATESUCCESSION, COEXISTENCE, NOTCOEXISTENCE
Intra-object	ATLEASTONE, ABSENCE, EXACTLYONE, RESPONDEDEXISTENCE, PRECEDENCE, ALTERNATEPRECEDENCE, RESPONSE, ALTERNATERESPONSE, SUCCESSION, ALTERNATESUCCESSION, COEXISTENCE, NOTCOEXISTENCE
Role	ABSENCE

only contain (activities related to) an *invoice* if there is also at least one activity related to a *delivery*, i.e., RESPONDEDEXISTENCE(*invoice*, *delivery*). As shown in Table 7.2, we use the same DECLARE templates to express inter-object constraints as we use for activity-level constraints, with the difference that the constraint parameters here correspond to objects, rather than activities. Consequently, each inter-object constraint is defined as follows:

Definition 15 (Inter-object constraint) *An inter-object constraint $c_{\mathcal{M}} \in C_{\mathcal{M}}$ is a tuple $c_{\mathcal{M}} = (type, templ, ot_1, ot_2, support)$, with $c_{\mathcal{M}}.type = interobj$, $c_{\mathcal{M}}.templ$ the constraint's* DECLARE *template, $c_{\mathcal{M}}.ot_1$ and $c_{\mathcal{M}}.ot_2$ its object types, and $c_{\mathcal{M}}.support$ its support.*

Intra-object constraints. Intra-object constraints capture relations regarding the actions, i.e., state changes, that are applied to an object type, e.g., that an *order* must be *checked* before it is *approved*, i.e., PRECEDENCE(*check*, *approve*) | *order*. We use the vertical bar (|) to denote the constraint's object type, i.e., that both actions are applied to an *order*. Although it is also possible to represent such constraints at the activity level (e.g., using PRECEDENCE(*check order*, *approve order*)), explicitly capturing them for a object offers greater flexibility in generalizing across different object types. For instance, in this manner, constraints like PRECEDENCE(*check*, *approve*) can be more effectively extended to other contexts, such as *sales orders*. This is advantageous during the fitting of constraints for a specific event log, as we show in the *Constraint Selection* stage (Section 7.2.2).

In addition to capturing pair-wise behavioral relations between actions, we also consider situations where specific actions must or must not be performed for certain types of objects. These are captured through the unary ATLEASTONE, ABSENCE,

and EXACTLYONE templates, which can, e.g., capture that, if a trace contains an activity related to a *purchase order item*, that trace must contain an activity that creates that item, i.e., EXACTLYONE(*create*) | *purchase order item*.

Based on this, each intra-object constraint is defined as follows:

Definition 16 (Intra-object constraint) *An intra-object constraint $c_{\mathcal{M}} \in C_{\mathcal{M}}$ is a tuple $c_{\mathcal{M}} = (type, templ, object, arity, n_1, n_2, support)$, with $c_{\mathcal{M}}.type = intraobj$, $c_{\mathcal{M}}.templ$ the constraint's* DECLARE *template, $c_{\mathcal{M}}.object$ its object, and $c_{\mathcal{M}}.arity \in \{\text{unary}, \text{binary}\}$ its arity indicating whether $c_{\mathcal{M}}$ is a unary or binary constraint, according to $c_{\mathcal{M}}.templ$. Moreover, $c_{\mathcal{M}}.n_1$ corresponds to the constraint's first action, $c_{\mathcal{M}}.n_2$ represents its second action (set to $\perp$ if $c_{\mathcal{M}}.arity = unary$), and $c_{\mathcal{M}}.support$ pertains to its support.*

Role constraints. Finally, role constraints focus on the resource perspective, restricting the execution of activities to specific roles, i.e., who in an organization can perform a given process step. These constraints can be expressed using the ABSENCE template with an activation condition. For instance, a role constraint may capture that an *approve order* activity must be performed by an employee with a *manager* role, written as ABSENCE(*approve order*) | `role` $\neq$ *manager*, with `role` $\neq$ *manager* as the activation condition. Each role constraint is defined as follows:

Definition 17 (Role constraint) *A role constraint $c_{\mathcal{M}} \in C_{\mathcal{M}}$ is a tuple $c_{\mathcal{M}} = (type, templ, a, r, support)$, with $c_{\mathcal{M}}.type = role$, $c_{\mathcal{M}}.templ =$*ABSENCE*, $c_{\mathcal{M}}.a$ the constraint's activity, $c_{\mathcal{M}}.r$ the role in its activation condition, and $c_{\mathcal{M}}.support$ its support.*

Constraint Extraction

The goal of constraint extraction is to derive a set of constraints C_M from each process model $M \in \mathcal{M}$. Using the constraint types and corresponding templates introduced in the previous section, we base the extraction of constraints on the idea of declarative constraint mining from event logs [59], where each template is instantiated with all possible parameters (or parameter combinations) before they are checked against the traces for satisfaction. We apply the same idea to the set of execution sequences allowed by a model, rather than to traces from an event log, with the extraction procedure depending on the constraint type:

Activity-level constraints. We extract activity-level constraints from a model $M = (A, F, R, \texttt{performedBy})$ by first instantiating potential constraints using each template from Table 7.2 and each pair-wise combination of activities in A. Then we check which of these potential constraints is satisfied by all and activated by at

least one of the execution sequences in F. For instance, for the RESPONSE template, we check for each pair (a_1,a_2) $\in A \times A$ if $\mathbf{G}(a_1 \rightarrow \mathbf{F}\ a_2)$ (see Table 2.4 for the LTL formulas of DECLARE templates) holds for all $\pi \in F$ and—if so—set $c_{\mathcal{M}}.type = activity$, $c_{\mathcal{M}}.templ =$ RESPONSE, $c_{\mathcal{M}}.a_1 = a_1$, and $c_{\mathcal{M}}.a_2 = a_2$ and add $c_{\mathcal{M}}$ to C_M.

Inter-object constraints. We extract inter-object constraints by obtaining a set of allowed object sequences F_{obj} and then checking which constraints hold between pairs of objects according to F_{obj}.

To obtain F_{obj}, we project each (activity) execution sequence $\pi \in F$ to an object sequence π_{obj}, which we achieve using our semantic annotation approach (cf. Chapter 3) that allows us to obtain the object types (if any) from each activity in π. For instance, $\pi = \langle$*create order*, *check order*, *approve order*, *ship goods*$\rangle$, yields $\pi_{obj} = \langle$*order*, *order*, *order*, *goods*$\rangle$. Activities that do not relate to any object type are not considered in the projected trace. While uncommon, if an activity relates to multiple object types, these are considered as a single object type, assuming that in a reference process model, these are then consistently referenced together.

Afterwards, we instantiate constraint templates between possible pairs of objects (ot_1, ot_2) occurring in F_{obj} and check them against the sequences in F_{obj}, in the same manner as done for activity-level constraints. For instance, for the binary RESPONDEDEXISTENCE template, we check for each combination (ot_1,ot_2) with $ot_1, ot_2 \in OT$ if $\mathbf{F}\ ot_1 \rightarrow \mathbf{F}\ o_2$ holds for all $\pi_{obj} \in F_{obj}$ and—if so—set $c_{\mathcal{M}}.type = interobj$, $c_{\mathcal{M}}.templ =$ RESPONDEDEXISTENCE, $c_{\mathcal{M}}.ot_1 = ot_1$, and $c_{\mathcal{M}}.ot_2 = ot_2$ and add $c_{\mathcal{M}}$ to C_M.

Intra-object constraints. We extract intra-object constraints for each object type that is contained in the activities in A based on the actions that are applied to them.

To obtain the necessary information on object types and actions, we, therefore, again use the semantic-annotation approach (Chapter 3), as done for inter-object constraints. This time, we turn the output of that approach into a function `getObjectActionPairs`, which, given an activity a, returns a set of object-type-action pairs obtained from a, where each pair is given as a tuple (ot, n), indicating that action n is applied to object ot.[1] E.g., `getObjectActionPairs` (*approve purchase order*) $= \{$(*purchase order*, *approve*)$\}$. By applying the function to each activity $a \in A$, we obtain a set OT of distinct object types that appear in the activities in A and a set N_{ot} of actions per object type $ot \in OT$, which we use to create intra-object sequences.

As shown in Algorithm 5, we create a set of intra-object action sequences F_{ot} for each object type $ot \in OT$ as follows: For each $\pi \in F$, we check for each

[1] Note that, for clarity, we here focus on situations where each activity yields a single object-action pair, although our approach can also handle activities with multiple pairs, e.g., *receive and check document*.

activity a_i in π if a_i refers to ot and if so only retain the action applied to ot. For instance, for $\pi = \langle \textit{create order}, \textit{check order}, \textit{approve order}, \textit{ship goods} \rangle$, we get $\pi_{\textit{order}} = \langle \textit{create}, \textit{check}, \textit{approve} \rangle$.

Algorithm 5 Creating intra-object sequence sets.

Input F: set of finite execution sequences; OT: set of object types
Output res: set containing a set of action sequences per object type

1: $res \leftarrow \{F_{ot} \leftarrow \emptyset \mid ot \in OT\}$ ▷ `Initialize an empty set of action sequences for each object type`
2: **for** each object type $ot \in OT$ **do**
3: **for** each $\pi \in F$ **do**
4: $\pi_o t \leftarrow \langle\rangle$
5: **for** each activity a_i in $\pi = \langle a_1, \ldots, a_n \rangle$ **do**
6: $(ot_{a_i}, n_{a_i}) \leftarrow$ `getObjectActionPairs`(a_i)
7: **if** $ot = ot_{a_i}$ **then**
8: $\pi_{ot} \leftarrow \pi_{ot} + \langle n_{a_i} \rangle$ ▷ `Retain the action applied to` ot `and append to current sequence`
9: $F_{ot} \leftarrow F_{ot} \cup \{\pi_{ot}\}$
10: **return** res

Based on an intra-object sequence set F_{ot}, we can instantiate DECLARE templates with actions in N_{ot} in the same manner as done with activities for activity-level constraints. For instance, for PRECEDENCE, we check for each combination $(n_1, n_2) \in N_{ot} \times N_{ot}$ if $\mathbf{G}(n_2 \rightarrow \mathbf{O}\, n_1)$(cf. Table 2.4) holds for all $\pi_o \in F_{ot}$ and—if so—set $c_{\mathcal{M}}.type = intraobj$, $c_{\mathcal{M}}.templ =$ PRECEDENCE, $c_{\mathcal{M}}.arity =$ binary, $c_{\mathcal{M}}.n_1 = n_1$, $c_{\mathcal{M}}.n_2 = n_2$, and $c_{\mathcal{M}}.object =$ *order* and add $c_{\mathcal{M}}$ to C_M.

Role constraints. Finally, role constraints can be extracted independently of execution sequences from the mapping `performedBy` of a model M, which maps an activity to the role that is supposed to perform it (if any). These can thus be created by instantiating the ABSENCE template with a corresponding activation condition setting $c_{\mathcal{M}}.type = role$, $c_{\mathcal{M}}.templ =$ ABSENCE, $c_{\mathcal{M}}.a = a$, and $c_{\mathcal{M}}.q = q$ and adding $c_{\mathcal{M}}$ to C_M for each activity $a \in A$ that `performedBy` maps to a role $r \in R$.

Through this procedure, we obtain a set of constraints C_M per model $M \in \mathcal{M}$. Having a large collection of models and a considerable number of constraints extracted per model, we next set out to refine this collection of constraint sets.

Collection Refinement

Having extracted independent constraint sets $\{C_M \mid M \in \mathcal{M}\}$ per reference process model, we next establish a single refined constraint collection $C_{\mathcal{M}}$ aggregating

the individual constraint sets. To this end, we standardize equal (or very similar) constraints and omit redundant constraints that are subsumed by stronger ones.

Standardizing constraints. First, we standardize the actions of the constraints that we extracted to obtain a single, more concise set of constraints $C_{\mathcal{M}}$. For instance, we avoid that PRECEDENCE(*create invoice, approve invoice*) and PRECEDENCE(*invoice created, invoice approved*), are treated as distinct constraints. Therefore, we standardize the activities of activity-level and action(s) of intra-object constraints by turning past-tense actions, like *created* and *approved*, into the present-tense, i.e., *create* and *approve*. We also update the word ordering of activity-level constraints accordingly, e.g., *invoice created* becomes *create invoice*.

After this standardization step, we can then aggregate the constraints from different models, setting a constraint's support to reflect from how many process models in $\mathcal{M}$ the particular constraint was extracted.

Removing redundant constraints. Finally, we remove weaker constraints that are encompassed by stronger ones, because these weaker ones are then redundant. To this end, we make use of the *subsumption relation* between DECLARE constraints, following the approach by Di Ciccio et al. [57]. For instance, RESPONSE(*a, b*), which encompasses RESPONDEDEXISTENCE(*a, b*) ("every RESPONSE is also a RESPONDEDEXISTENCE"). The subsumption relations between templates employed by our approach are shown in Figure 7.3, where the subsumption relation is depicted as a line starting from the subsumed template with an empty triangular arrow and ending in the subsuming one. For each $c^w_{\mathcal{M}} \in C_{\mathcal{M}}$, we check if a stronger (subsuming) constraint $c^s_{\mathcal{M}}$ exists that was observed the same number of times, i.e., $c^w_{\mathcal{M}}.support = c^s_{\mathcal{M}}.support$, and—if so—remove $c^w_{\mathcal{M}}$ from $C_{\mathcal{M}}$.

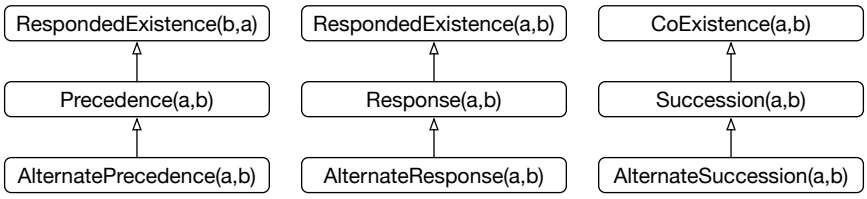


Figure 7.3: Subsumption relations between constraint templates employed by our approach (adapted from [57]).

Output of the mining stage. The output of this stage is a refined collection of mined constraints $C_{\mathcal{M}}$, which together with an event log L serves as the input of the next stage, the *Constraint Selection* stage.

7.2.2 Constraint Selection

$C_{\mathcal{M}}$ is based on the the entire reference model collection and therefore might contain various irrelevant constraints for a concrete event log L. The *Constraint Selection* stage therefore aims to select those constraints from $C_{\mathcal{M}}$ that are most relevant for the detection of best-practice violations in a given event log L. This involves three steps, as illustrated in Figure 7.4. In the first step, we "fit" the constraints for L, which yields a collection of fitted constraints C_L (we will explain what is meant by "fitting" below). Out of these, we recommend the most relevant ones C_L^R, taking into account selected configuration parameters and additional user preferences. Finally, we check the set of selected constraints C_L^R for consistency to ensure they are without any contradictions, which yields a consistent set of selected, fitted constraints C_L^S. In the following, we explain these steps in detail.

Figure 7.4: Overview of the constraint selection stage.

Constraint Fitting

In this step, our approach tries to fit the mined constraints in $C_{\mathcal{M}}$ to the contents of the given event log L, by relating the activities, objects, actions, and roles used in mined constraints (derived from models in $\mathcal{M}$), to counterparts found in the event log L, yielding a set of fitted constraints C_L. To do this, our approach looks for semantically similar counterparts, allowing it to, e.g., relate a PRECEDENCE(*receive invoice*, *pay invoice*) constraint to *receive bill* and *make payment* activities found in a log. Each such fitted constraint is defined as follows:

Definition 18 (Fitted constraint) *Given a mined constraint $c_{\mathcal{M}}$ and an event log L, a fitted constraint c_L is a tuple $c_L = (c_{\mathcal{M}}, type, templ, [components], support, sim, relevance)$, with c_L's type, templ, and support being equal to those of $c_{\mathcal{M}}$. $c_L.[components]$ is a shorthand to refer to the list of c_L's type-specific components (e.g., $c_L.[components] = c_L.a_1, c_L.a_2$ for activity-level constraints) that are each linked to components (i.e., activities, actions, objects, or roles) found in L. Finally, $c_L.sim$ is a map capturing the semantic similarity between $c_{\mathcal{M}}$'s and c_L's components, and $c_L.relevance$ is a score indicating the relevance of c_L for the event log L.*

Creating fitted constraints for L involves matching constraint components to their event log counterparts and replacing these components with their counterparts, as

shown by Algorithm 6. The algorithm takes as input the constraint collection $C_{\mathcal{M}}$ and the event log L, using a type-specific fitting procedure for each $c_{\mathcal{M}} \in C_{\mathcal{M}}$.

Algorithm 6 Constraint fitting.

Input $C_{\mathcal{M}}$: set of mined constraints; L: event log
Output C_L: set of fitted constraints

$C_L \leftarrow \emptyset$
for $c_{\mathcal{M}} \in C_{\mathcal{M}}$ **do**
  if $c_{\mathcal{M}}.type$ is $activity$ **then**
    for $(a_{1_L}, a_{2_L}) \in A_L \times A_L$ **do**
      if $\texttt{match}(c_{\mathcal{M}}.a_1, a_{1_L}) \wedge \texttt{match}(c_{\mathcal{M}}.a_2, a_{2_L}) \wedge a_{1_L} \neq a_{2_L}$ **then**
        FitActivityConstraint($c_{\mathcal{M}}, a_{1_L}, a_{2_L}$)
  if $c_{\mathcal{M}}.type$ is $interobj$ **then**
    for $(ot_{1_L}, ot_{2_L}) \in OT_L \times OT_L$ **do**
      if $\texttt{match}(c_{\mathcal{M}}.ot_1, ot_{1_L}) \wedge \texttt{match}(c_{\mathcal{M}}.ot_2, ot_{2_L}) \wedge ot_{1_L} \neq ot_{2_L}$ **then**
        FitInterObjectConstraint($c_{\mathcal{M}}, ot_{1_L}, ot_{2_L}$)
  if $c_{\mathcal{M}}.type$ is $intraobj$ **then**
    for $ot_L \in OT_L$ **do**
      if $\texttt{match}(c_{\mathcal{M}}.object, ot_L)$ **then**
        if $c_{\mathcal{M}}.arity =$ unary **then**
          for $n_L \in N_{o_L}$ **do**
            if $\texttt{syn}(c_{\mathcal{M}}.n_1, n_L)$ **then**
              FitIntraObjectConstraint($c_{\mathcal{M}}, n_L, \perp$)
        if $c_{\mathcal{M}}.arity =$ binary **then**
          for $(n_{1_L}, n_{2_L}) \in N_{ot_L} \times N_{ot_L}$ **do**
            if $\texttt{syn}(c_{\mathcal{M}}.n_1, n_{1_L}) \wedge \texttt{syn}(c_{\mathcal{M}}.n_2, n_{2_L}) \wedge n_{1_L} \neq n_{2_L}$ **then**
              FitIntraObjectConstraint($c_{\mathcal{M}}, n_{1_L}, n_{2_L}$)
  if $c_{\mathcal{M}}.type$ is $role$ **then**
    for $a_L \in A_L$ **do**
      if $\texttt{match}(c_{\mathcal{M}}.a, a_L)$ **then**
        for $r_L \in R_L$ **do**
          if $\texttt{match}(c_{\mathcal{M}}.r, r_L)$ **then**
            FitRoleConstraint($c_{\mathcal{M}}, a_L, r_L$)
return C_L

Semantic similarity computation. A key part of Algorithm 6 is the match($x_{\mathcal{M}}$, x_L) function, which is used to decide whether a constraint component $x_{\mathcal{M}}$ is semantically similar to an event log component x_L. It is instantiated using a similarity measure sim and a threshold ϵ, i.e., match($x_{\mathcal{M}}$, x_L):= $\texttt{sim}(x_{\mathcal{M}}, x_L) > \epsilon$. As a similarity measure, we employ the cosine similarity based on embedding vectors for $x_{\mathcal{M}}$ and x_L, which we obtain from a pre-trained *sentence transformer* [161].

These transformers are specifically designed to capture semantically meaningful representations on the level of sentences rather than individual words and can thus be applied on activities or object types that may consist of multiple words. We set ϵ to 0.5 by default, ensuring that all fitted constraints are reasonably similar to the components of L, which avoids having to filter out a lot of constraints from C_L^S in the next step. Note that, in order to make `match` more or less restrictive, it is possible to use different values for ϵ depending on the type of component we aim to find matches for, i.e., setting different thresholds for activities, objects, and actions.

Activity-level constraints. For an activity-level constraint $c_{\mathcal{M}}$, we need to find activities in L that correspond to the constraint's activity components, $c_{\mathcal{M}}.a_1$ and $c_{\mathcal{M}}.a_2$. Specifically, we create a fitted version of $c_{\mathcal{M}}$ for any pair of distinct log activities a_{1_L} and a_{2_L} for which both $\texttt{match}(c_{\mathcal{M}}.a_1, a_{1_L})$ and $\texttt{match}(c_{\mathcal{M}}.a_2, a_{2_L})$ hold (lines 5–6).

It is important to note that this can result in multiple fitted constraints being added to C_L that stem from a single mined constraint $c_{\mathcal{M}}$. To illustrate this, consider a mined constraint that specifies that requests must first be examined before a decision can be made on them, i.e., PRECEDENCE(*examine request*, *decide on request*). It is well-imaginable that a process like this actually uses different kinds of examinations for different types of cases or situations, e.g., having *perform basic examination* and *perform thorough examination*. In such a scenario, the mined constraint needs to be fitted twice, so that both types of examinations are covered, yielding constraints capturing PRECEDENCE(*perform simple examination*, *decide on request*) and PRECEDENCE(*perform thorough examination*, *decide on request*).

Inter-object constraints. To be able to fit an inter-object constraint $c_{\mathcal{M}}$, we need to find two object types in L that match the object types of $c_{\mathcal{M}}$. To do this, we use OT_L to denote the set of object types in L, which is obtained using `getObjectActionPairs` (see Section 7.2.1), i.e., $OT_L = \{ot \mid (ot, n) \in \{\texttt{getObjectActionPairs}(a) \mid a \in A_L\}\}$. Then, the fitting procedure for inter-object constraints is identical to the procedure for activity-level constraints, with the exception that we match the constraint's objects to objects in OT_L (lines 9–10).

Intra-object constraints. For an intra-object constraint $c_{\mathcal{M}}$, we need to find both a matching object type and one or two corresponding actions (depending on if $c_{\mathcal{M}}$ is a binary or unary constraint) in L. To this end, we first check for a matching object type ot_L, in the same manner as done for inter-object constraints (line 13). If found, we next check for matches between $c_{\mathcal{M}}$'s action(s) and those in the log that are applied to ot_L, i.e., in the set N_{ot_L} (see Section 7.2.1).

A distinction we make here is that we do not look for corresponding actions based on semantic similarity (i.e., using `match`), but rather look for synonyms of actions. We make this distinction because actions generally correspond to verbs,

which is a specific class of words that is covered well by lexical resources, such as *WordNet* [131], which provide curated lists of similar terms (e.g., synonyms). Therefore, by checking for synonymous actions, we can use a restrictive, but highly precise matching strategy for actions. Note that this synonym-based matching strategy using `syn` can be straightforwardly replaced with `match` if desired.

For instance, consider a constraint $c_{\mathcal{M}}$ corresponding to ATLEASTONE(*check*) | *order* and an event log that relates to *purchase order* objects, including an *examine purchase order* activity. Then, our approach would recognize that *order* and *purchase order* are semantically similar, i.e., `match`(*order*, *purchase order*)$= true$, and that *examine* is a synonym of *check*, i.e., `syn`(*check*, *examine*)$= true$, yielding a fitted constraint c_L corresponding to ATLEASTONE(*examine*) | *purchase order*.

Role constraints. Finally, to fit a role constraint $c_{\mathcal{M}}$, we need to identify both a matching activity and a matching role in L. To this end, we first check for a matching activity $a_L \in A_L$ in the same manner as for activity-level constraints (line 24). If found, we next check for matches between $c_{\mathcal{M}}$'s role $c_{\mathcal{M}}.r$ (captured in its activation condition) and the roles in the log, i.e., in the set R_L. If for $r_L \in R_L$, `match`($c_{\mathcal{M}}.r$, $r_L = true$) holds, we fit $c_{\mathcal{M}}$ with a_L and r_L (line 27).

When fitting a constraint, we populate c_L's similarity map $c_L.sim$ such that it maps the matched components from $c_{\mathcal{M}}$ and their counterparts in c_L to their respective similarity scores, which we will use for recommendation. Having iterated through all constraints $c_{\mathcal{M}} \in C_{\mathcal{M}}$, the algorithm outputs the set of fitted constraints C_L.

Constraint Recommendation

While from the previous step we have obtained a set of fitted constraints, this set might still be too large, or contain constraints that are uninteresting. In the recommendation step, we therefore aim to select a subset of constraints from C_L that are most relevant to the event log L. To this end, we first compute a relevance score for each constraint, which we then use to select a set C_L^R of recommended constraints, according to user-defined parameters regarding size and relevance.

Relevance computation. To be able to recommend appropriate constraints, we need an indicator of how relevant a constraint $c_L \in C_L$ is to an event log L. To achieve this, we compute the relevance score $c_L.relevance$ based on c_L's semantic similarity scores and support. We consider these two aspects, since a higher semantic similarity indicates that a constraint is more applicable to the specific context of the given event log L (e.g., because the activities from the original, mined constraint are highly similar to those found in L), whereas higher support indicates that a constraint is more generally applicable to processes (because it was found in a larger number of models in $\mathcal{M}$). We capture these aspects as follows:

$$c_L.relevance = \omega \cdot \mathsf{avg}(\{s \mid ((x,y),s) \in c_L.sim\}) + (1-\omega) \cdot \frac{c_L.support}{\mathsf{max}(\{c'_L.support \mid c'_L \in C_L, c'_L.type = c_L.type\})} \quad (7.1)$$

As shown, we combine the average similarity score $\mathsf{avg}(\{s \mid ((x,y),s) \in c_L.sim\})$ observed for the constraint with the constraint's support $c_L.support$, normalized by dividing it by the maximum support of a constraint of the same type in C_L, such that both, similarity and support, have a comparable range. The semantic similarity and support parts can be weighed differently by adapting $\omega \in [0,1]$, e.g., to give a higher weight to support.

Constraint selection. The computed relevance scores can be employed to obtain a selection of recommended constraints C_L^R. Our approach allows users to select all constraints from C_L that have a relevance higher than a certain threshold τ, or to select the top-k constraints from C_L that have the highest relevance scores, thus providing users control over either the minimum desired relevance score τ or the maximal number of constraints k. Users can also apply the selection strategies in a type-specific manner, i.e., by specifying different relevance thresholds τ_{type} for each constraint type, or by selecting the top-k constraints per type.

User input. After providing a set of recommended constraints C_L, the user can decide to continue without intervention or inspect the recommended constraints and optionally filter the collection manually. For instance, if they find that some recommended constraints involve an object that is not interesting from a compliance perspective, they can remove all constraints that involve that object.

Consistency Checking

The final step of the *Constraint Selection* stage ensures that C_L^R contains no inconsistencies. For instance, we want to avoid that the set contains both CoExistence(a,b) and NotCoExistence(a,b) constraints, since these directly contradict each other. Note that inconsistencies can also be more complex due to transitivities, e.g., a set of constraints Response(a,b), Response(b,c) and NotCoExistence(a,c) is also inconsistent, since the first two constraints transitively state that a should eventually be followed by c, whereas the final constraint explicitly forbids this. Since such combinations are hard, if not impossible, to spot for humans [133], automated consistency checking and resolution is required.

To operationalize this, we use an existing approach for inconsistency resolution in declarative process specifications [48]. Given a set of constraints C, this

approach identifies a set of minimal correction sets $\mathsf{MCS}(C)$, which are sets of constraints that, if removed from C, make C consistent:

Definition 19 (Minimal correction sets) *Given a constraint set C, a set $X \subseteq C$ is a minimal correction set of C, if $C \setminus X$ is consistent, and $\forall X' \subset \Phi : C \setminus X'$ is inconsistent. We denote $\mathsf{MCS}(C)$ as the set of minimal correction sets for the constraints in C.*

If the inconsistency resolution approach detects inconsistencies in C_L^R, our approach, by default, deletes the constraints from the correction set in $\mathsf{MCS}(C_L^R)$ with the lowest cumulative relevance score. This ensures that we obtain a consistent set C_L^S of constraints, while maximizing the overall relevance of the retained constraints. If there are no inconsistencies, C_L^R simply becomes C_L^S.

User input. The inconsistency-resolution approach [48] also allows a user to select specific constraints that should not be removed from C_L^R, giving them control over the recommended correction sets. Furthermore, users have the option to inspect the correction sets and choose to remove one of these based on their preferences instead of automatically having the one removed that maximizes relevance.

Output of the selection stage. The output of this stage is a collection of fitted constraints C_L^S that is free of inconsistencies. C_L^S is used in the subsequent stage to check for best-practice violations in traces of the event log L.

7.2.3 Conformance Checking

Our approach's final stage uses the set of selected constraints C_L^S to check for best-practice violations in the event log L and, if found, presents these to the user. To this end, the approach first identifies violations per trace, yielding a trace-based violation map V_L. Afterwards, it aggregates the violations in V_L to the log-level and adds an explanation of each type of violation, providing a summary to the user.

Trace-Based Conformance Checking

The goal of this step is to check which constraints are violated per trace $\sigma \in L$. To this end, our approach first creates a collection of trace-constraint pairs DV_L, where each pair represents the detected violation of c_L in σ. It then uses DV_L to create a map V_L that assigns each trace the best-practices that it violates.

To establish a collection DV_L of trace-constraint pairs (σ,c_L), our approach checks per trace $\sigma \in L$ and each constraint $c_L \in C_L^S$ if that constraint is violated, i.e., if $\sigma \not\models c_L$. The checking process is analogous to the one used in constraint mining (cf. Section 7.2.1). However, while in constraint mining we assess whether

a constraint is fulfilled across all sequences allowed by a model, here we focus on identifying specific traces where a constraint is not met. Given a trace σ, the conformance checks per constraint type are performed as follows:

Activity-level and role constraints. For an activity-level or a role constraint c_L, our approach can directly check if $\sigma \not\models c_L$ holds and, if so, add (σ, c_L) to DV_L.

Inter-object constraints. For an inter-object constraint c_L, our approach checks against the object sequence σ^a_{obj} that it obtains by projecting the activity sequence σ^a to its objects, as also done when mining constraints of this type. Then, it checks if $\sigma^a_{obj} \not\models c_L$ holds and, if so, adds (σ, c_L) to DV_L.

Intra-object constraints. For an intra-object constraint c_L, the approach checks against the action sequence σ_o for its object type $ot = c_L.object$, which it obtains in the same manner as done in Algorithm 5. Then, it checks if $\sigma_o \not\models c_L$ holds and, if so, adds (σ, c_L) to DV_L.

To report on best-practice violations per trace, our approach establishes a map V_L based on the pairs in DV_L, which associates each trace σ in L with a set of constraints that it violates (or an empty set if σ does not violate any constraints).

Result Summarization

In this final step, our approach creates a more concise and informative summary of the detected best-practice violations for the user by aggregating them across traces. Specifically, it groups violations per constraint type and template and adds a textual explanation to each such violation. An example of this is shown in Table 7.3. To generate the explanations for the violations, we use a standard sentence template for each combination of constraint type and DECLARE template.

Constraint	**Constraint explanation**	**Affected traces**
ATLEASTONE (*create*)|*order*	Each *order* must be *created*	$\{\sigma_6, \ldots, \sigma_{76}\}$
PRECEDENCE (*check*,*approve*)|*invoice*	An *invoice* must be *checked* before it is *approved*	$\{\sigma_{23}\}$

Table 7.3: Exemplary result summary of intra-object violations.

Output of the conformance-checking stage. The output of this final stage consists of (1) a map V_L assigning each trace in L the best-practices that it violates and (2) a violation summary providing an overview of best-practice violations contained in L, explicating the traces that are affected by them.

7.3 Evaluation

In this section, we aim to demonstrate the effectiveness of our approach to select relevant constraints for detecting best-practice violations. To do so, we test if the constraints that our approach selects for a given log can be used to find known violations. We describe the data collection in Section 7.3.1 and the experimental setup in Section 7.3.2. In Section 7.3.3, we present our evaluation results demonstrating our approach's efficacy in detecting best-practice violations and that it outperforms a state-of-the-art approach in both scope and accuracy. The implementation, data collection, evaluation pipeline, and raw results are all available in our repository[2].

7.3.1 Data Collection

Our goal is to show that our approach is effective in selecting constraints that are relevant for a given event log and that these constraints can be used to detect best-practice violations. Therefore, we need a process model collection to extract constraints from and event logs with known best-practice violations, i.e., with behavior that deviates from a process model that represents desired behavior.

Process model collection. Since there is no public reference model collection available, we use a collection of real-world process models. We use this collection for both extracting constraints and generating event logs with best-practice violations. For the experiments, we apply a cross-validation procedure as explained later. Specifically, we use the public *SAP-SAM* data set [167], a large process model collection created by academic users of a commercial process modeling tool.

Given that there is no quality assurance for the entire model collection, we select only English models that fulfill a set of requirements to reduce data quality issues. In particular, each model needs to have between 5 and 50 elements, pass the BPMN syntax check (using the functionality of a commercial process modeling tool), and must be transformable into a sound workflow net. The former two requirements reduce the probability that models with barely any (too few elements) or pointless (too many elements and syntax errors) behavior are included, whereas the latter ensures that we can generate proper event logs from the models. From those we randomly select 1,500 models, primarily in order to keep run times manageable. Note that because this is not a real *reference* process model collection, we assume that the behavior of the contained models captures best practices.

Generating event logs with best-practice violations. To obtain logs with known violations from this process model collection, we play out the models and insert noise: For each model, we use its workflow net to generate an event log that con-

[2] https://github.com/a-rebmann/semantic-constraint-miner/

tains a single trace of each variant that is possible in the net (in case of loops, each loop is executed at most once). For nets with fewer than 100 variants, we keep playing out traces until the log has 100 traces, so that we have a minimum amount of traces available per log for noise insertion. Then, to add violations to the logs, we select traces to introduce noise into, with a probability of 50% per trace. If a trace is chosen for noise insertion, we either randomly add, remove, or swap events or assign a different role to an event than the original one. After performing a noise-insertion action for a selected trace, there is a 50% probability to insert noise again (repeated until false). The characteristics of the log collection (before and after noise insertion) obtained in this manner are shown in Table 7.4. As depicted there, the complexity of the logs varies considerably. For instance, the logs have 5 activities on average, whereas the maximum number observed is 18 and the number of object types is 3.9 on average, whereas the maximum is 17. The differences between the original and noisy logs become clear when looking at the variants. While the original logs contain 3.5 different variants on average, this increases to 30.9 for the noisy logs.

Table 7.4: Characteristics of the original and noisy logs.

Collection	**Count**	**Activities**		**Object types**		**Variants**		**Variant length**	
		avg.	**max.**	**avg.**	**max.**	**avg.**	**max.**	**avg.**	**max.**
Original logs	1,500	4.9	18	3.9	17	3.5	720	3.7	10
Noisy logs	1,500	4.8	18	3.9	17	30.9	703	3.9	14

The inserted noise then leads to violations of activity, inter-object, intra-object, and role constraints. Because we assume that a process model established by users contains semantically sound behavior, any behavior not allowed by the model can be considered a best-practice violation. Hence, we compute the known best-practice violations for an event log L by verifying for each trace $\sigma \in L$ if σ violates the behavior of the model M that was used to generate L and recording violations of this behavior in a collection of true best-practice violations L_V^M. Statistics about the best-practice violations contained in our log collection are depicted in Table 7.5. As shown there, on average 40% of traces are affected by activity-level violations, 28% by inter-object violations, 38% by intra-object violations, and 9% by role violations. Across constraint types, the average number of violations per log is considerably higher than the number of affected traces, indicating that frequently there are multiple violations per trace.

Table 7.5: Violations per noisy log.

Const. type	Affected traces		Violations	
	avg.	max.	avg.	max.
Activity	40.77	312	169.00	1979
Inter-object	28.34	276	95.35	1453
Intra-object	38.31	199	91.22	905
Role	9.14	150	11.47	172

7.3.2 Evaluation Setup

Implementation. We implemented our approach and evaluation pipeline in Python. For handling the import and generation of event logs, we used *PM4Py* [32]; as a basis for constraint checking, we used *Declare4Py* [63]. To assess semantic similarity between activities and business objects, we generated sentence transformer embeddings [161] using the *all-MiniLM-L6-v2* pretrained language model[3]. Finally, as described in Section 7.2, we use our semantic-annotation approach (Section 3.3) to extract objects and actions from activities and the approach by Corea et al. to check for inconsistencies in constraint sets [48].

Configurations. We test various settings for the relevance threshold τ, the weight of the semantic similarity ω when computing the relevance of a constraint, and k that determines how many of the most relevant constraints per type are selected. In particular, we run experiments for $\tau \in \{0.5, 0.8\}$ to investigate the effect of a low vs. a high relevance score, $\omega \in \{0.5, 0.9\}$ to investigate the importance of support, and $k \in \{10, 100, 250\}$ to investigate how the number of selected constraints impacts the performance of our approach.

Baseline. We compare our approach to the state-of-the art approach for semantic anomaly detection proposed by Van der Aa et al. [5]. The approach populates a knowledge base with semantic relations that reflect appropriate process executions. The knowledge records for this population procedure are extracted from (1) a general-purpose linguistic resource that captures relations between verbs and (2) a process model collection (i.e., the same process model collection we use to mine constraints from). For a given trace, the approach checks whether pairs of actions that are applied to the same object violate records in the knowledge base, indicating semantic anomalies. We compare our work against the configuration with the best reported results in the original paper, referred to as *SEM4*. Importantly, this configuration leverages semantic similarity to enhance the generalizability of the rules stored in the knowledge base. Note that, because of its focus on actions applied to the same object, the baseline can solely detect intra-object violations.

[3] Available here: `https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2`

Cross validation. Because we use the same model collection $\mathcal{M}$ for constraint mining and detection, we conduct a 5-fold cross validation. To this end, we randomly split the model collection into 5 sets and use models of 4 sets for constraint mining and the noisy logs generated from the 5th set for testing the violation detection. We run the experiments 5 times, so that each set acts as the test set once.

Measures. We check for best-practice violations in the noisy log L^M, given the original model M, which yields a collection of detected violations L_D^M. For each detected violation $v \in L_D^M$, we check if v is contained in the set of true violations L_V^M (computed as explained above) resulting in a true positive (TP) or false positive assessment (FP). Any violation that is present in L_V^M but has not been detected by our approach, i.e., is not included in L_D^M, is deemed a false negative (FN). Using this approach, we quantify the precision, given by TP/(TP+FP) and the recall, given by TP/(TP+FN). We also report on run time for the different stages.

7.3.3 Evaluation Results

We first report on the constraints mined from the model collection, before describing the overall results of the best-practice-violation detection. Then, we report on the results per constraint type and inspect the detailed results of selected logs to provide insights into when our approach performs well and when it fails. Finally, we compare our approach against the baseline and reflect on its run time.

Extracted constraints. Our approach extracted more than 250,000 constraints from the 1,500 process models. As shown in Table 7.6, this includes more than 150,000 activity-level, almost 60,000 inter-object, more than 42,000 intra-object constraints, and more than 3,000 role constraints. Per process model, it extracted at most 662 activity-level constraints (104.0 on average), 416 inter-object constraints (54.1 on average), 366 intra-object constraints (33.3 on average), and 17 role constraints (6,3 on average).

Table 7.6: Characteristics of the constraints extracted from the model collection before collection refinement

Const. type	Count	Avg. per model	Min./Max. per model
Activity	150,082	104.0	2 / 662
Inter-object	59,452	54.1	2 / 416
Intra-object	42,377	33.3	3 / 366
Role	3,349	6.3	0 / 17

After refinement, i.e., after equal or highly similar constraints were standardized and weaker, redundant ones were filtered out, 55,913 activity-level, 20,920 inter-object, 8,223 intra-object constraints, and 2,387 role constraints remain, as

depicted in Table 7.7. Many constraints were extracted from multiple models, as shown by the number of constraints with a support of > 1. Notably, more than half of the intra-object constraints (4,484) were extracted from multiple models, which suggests that these are more generally applicable than the other types of constraints, for which this ratio is lower.

Table 7.7: Characteristics of the constraint collection once aggregated and refined.

Const. type	Count	Support > 1
Activity	55,931	5,277
Inter-object	20,920	2,965
Intra-object	8,223	4,484
Role	2,387	266

Overall detection results. Table 7.8 depicts the evaluation results per configuration for detecting best-practice violations from the noisy logs, averaged across these. We generally favor high recall over high precision and the best trade-off between recall and precision per constraint type is printed in bold. In the table, k determines the number of most relevant constraints selected per type. τ is a threshold for the minimum relevance score that a constraint must have to be selected. The weight factor of the relevance score, ω, is set to 0.9 in all configurations. Our experiments showed that the performance trends per configuration are the same across folds. We also found that weighing the semantic relevance ω with 0.9 for all constraint types consistently yields slightly better results than weighing support equally. For the other parameters the variety of the performance achieved is more substantial, though. Therefore, we report on results with $\omega = 0.9$ here for brevity.[4]

Overall, we find that both precision and recall vary considerably across configurations (0.37–0.86 resp. 0.35–0.85). Because we favor high recall over high precision when detecting best-practice violations, the best out of the tested configurations, which achieves good recall while maximizing precision is $k = 100$ combined with a low relevance threshold ($\tau = 0.5$) for all constraint types. By picking this configuration, we achieve an average precision of 0.50 and a recall of 0.81 across constraint types. Note that, while this configuration achieves the best results across constraint types, we do not need to use the same configuration per type. Selecting a high relevance threshold ($\tau = 0.8$) yields considerably lower recall scores (0.35–0.65) compared to when using a lower threshold ($\tau = 0.5$; 0.57–0.85). This indicates that applying such a high relevance threshold is too restrictive, causing a considerable amount of best-practice violations to remain undetected.

[4]Detailed results including all configurations can be found in our repository.

Table 7.8: Results of detecting best-practice violations per constraint type for various configurations (averaged across logs).

Type	Config. k	Config. τ	TP	FP	FN	Precision	Recall
Activity	10	0.5	77.10	62.04	91.90	0.70	0.51
	100	0.5	134.22	230.83	34.78	**0.43**	**0.82**
	250	0.5	139.80	286.17	29.20	0.39	0.84
	10	0.8	54.54	24.82	114.46	0.86	0.35
	100	0.8	80.08	76.61	88.92	0.75	0.49
	250	0.8	80.27	79.64	88.73	0.74	0.49
Inter-object	10	0.5	50.56	77.46	44.79	0.59	0.58
	100	0.5	77.57	231.39	17.77	**0.34**	**0.82**
	250	0.5	79.47	277.79	15.88	0.32	0.84
	10	0.8	42.07	52.11	53.28	0.70	0.48
	100	0.8	53.87	98.07	41.48	0.59	0.59
	250	0.8	53.87	98.33	41.48	0.59	0.59
Intra-object	10	0.5	51.69	26.46	39.53	0.84	0.70
	100	0.5	68.64	131.71	22.58	**0.72**	**0.79**
	250	0.5	69.55	172.97	21.67	0.70	0.79
	10	0.8	46.74	21.26	44.48	0.86	0.59
	100	0.8	58.47	94.93	32.75	0.78	0.65
	250	0.8	58.80	104.46	32.42	0.77	0.65
Role	10	0.5	6.51	16.73	4.96	0.60	0.57
	100	0.5	9.01	37.39	2.46	**0.49**	**0.80**
	250	0.5	9.08	38.31	2.39	0.49	0.80
	10	0.8	4.39	8.47	7.08	0.78	0.38
	100	0.8	4.77	10.78	6.69	0.76	0.41
	250	0.8	4.77	10.78	6.69	0.76	0.41
Overall	100	0.5	72,36	157,82	19.40	0.50	0.81

We also looked into per-log performance to get insights into how often our approach achieves excellent, good, and poor performance. Figure 7.5 shows the number of logs for which various performance levels are reached, using the best configuration per constraint type ($k = 100$, $\tau = 0.5$). We find that our approach achieved perfect precision and recall for a considerable number of logs. Especially for intra-object constraints this was the case for more than 350 logs. For the others the approach does not perform as well but still for the vast majority, i.e., more than 90% of the logs, a recall (the most important score) of more than 0.5 was achieved. Only for very few logs both precision and recall are poor, i.e., both below 0.25.

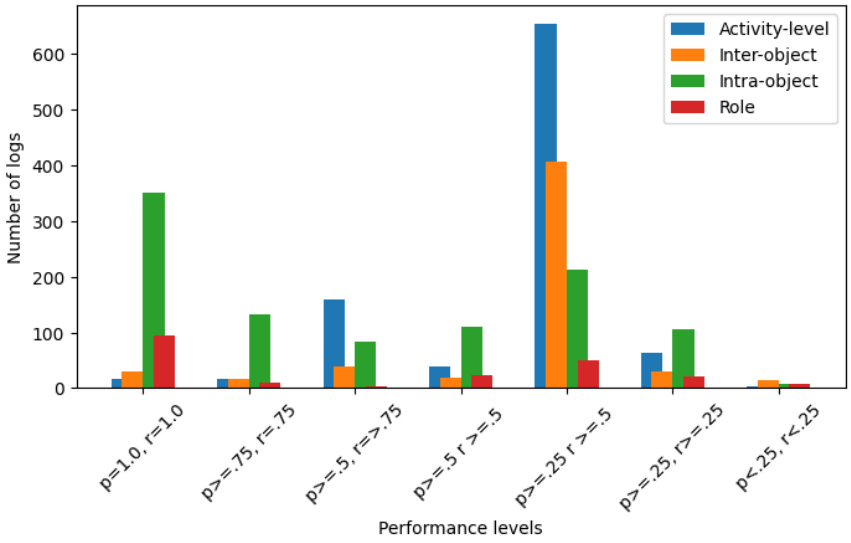


Figure 7.5: Number of logs for different performance levels per constraint type (considering logs for which there were best-practice violations to be detected for the given type of constraint); p=precision, r=recall, lower performance levels exclude logs that are already included in higher ones.

Results per constraint type. To get a better understanding of when the approach excels and when it fails, we next discuss the results per constraint type by analyzing the impact of the parameter settings on performance and inspecting the results achieved for individual event logs.

Activity-level constraints. The best configuration achieves a moderate precision of 0.43 and a good recall of 0.82 for activity-level constraints. We find that we primarily obtain perfect scores for rather standard types of processes, where organization-specific deviations are expected to occur rather rarely. For instance, for an order-handling event log, all 246 best-practice violations were correctly identified without any false positives, recognizing, e.g., that *confirm order* must precede *emit invoice*, that *archive order* should only happen after *receive payment*, and that *ship product* should not occur before *get shipment address* if the latter occurs. The same applies to a loan-application log, where for instance, *finalize loan application* must naturally not precede *create application*. Conversely, we find that logs for which our approach achieved poor performance correspond to rather specialized processes, for instance, for a log of a process that handles the calculation of compensation during parental leave, our approach only achieved a precision of 0.30 and recall of 0.56 and for a log that captures the interaction between a technician and a pharmacist with a software system, we achieved a precision of 0.20 and re-

call of 0.78. This indicates that activity-level constraints are especially useful to detect best-practice violations in processes that are executed by many organizations in similar ways, i.e., the main use case for reference process models which our approach expects as input.

Inter-object constraints. For inter-object constraint, we observe similar trends as for activity-level ones. The best configuration achieves a precision of 0.34 and a recall of 0.82. For rather streamlined processes, such as a claim-handling process, it correctly detects that a *claim* should occur before a *payment* and that *payment* and *claim rejection* cannot co-occur. Accordingly, in a travel booking process, it perfectly identifies that a *plane ticket* should be involved in a process before *luggage* is and a *destination* should occur before there is a *plane ticket*. Larger numbers of false positives are detected, for instance, when the process contains many different object types and at the same time there is a lot of variability in the process, i.e., many execution variants. For instance, in a log capturing a heath-care process that spans multiple departments, 349 false positive violations were found, compared to 75 true positive ones.

Intra-object constraints. When considering intra-object constraints, we observe substantially better performance compared to the other constraint types. The best configuration achieves a precision of 0.72 and a recall of 0.79. For more than 400 logs a perfect recall and a precision above 0.75 was obtained at the same time. The reason for the better results compared to other constraint types can also be observed by inspecting results of individual logs. For instance, we achieve perfect recall and precision for a sales process, for which, e.g., best-practice violations were found where an order was archived before it was rejected, an order was both rejected and confirmed, and an order was confirmed before it was received. Similarly, in a credit-application log we found credits that are offered before they are checked and applications that are never assessed. These examples show that our approach is capable of identifying violations of general behavioral relations that make sense, such as that an object should be created before it is checked.

However, there are also cases for which our approach fails for intra-object violation detection, i.e., misses true best-practice violations. For instance, in an event log that captures the process of candidate selection for scholarships. In the process, candidates are first classified and later selected. Our approach failed to recognize best-practice violations where one of these were missing or their order was swapped causing false negatives. A possible reason for this may be the similar meaning of the actions *select* and *classify*, which both describe the separation of elements of a group [122]. Still, we found that intra-object constraints provide a reliable means to detect best-practice violations involving the same type of object.

Role constraints. For role constraints, the best configuration achieves a precision of 0.49 and recall of 0.8. We also inspected logs for which role constraints performed

perfectly. For instance, in a credit-check process all best-practice violations of wrong roles performing a task were correctly identified, e.g., that calling a client should be performed by *customer contact*, whereas a credibility check should be performed by *risk management* and a payment should be issued by *finance*. However, we also observed false positives. For instance, for a log capturing the interaction of a pharmacist with an information system that performs some tasks autonomously, such as sending and processing notifications, the approach selected constraints that did not correctly identify who (i.e., the pharmacist or the system) should perform which steps of the process.

Main insights. Overall, we observe that with a relatively small number of constraints, picked from a large collection of thousands of constraints per type, we detect a substantial amount of true best-practice violations, achieving good recall scores across constraint types. Yet, there are also false positives, which in turn lead to lower precision scores. For the goal of detecting best-practice violations allowing for false positives while minimizing false negatives can be acceptable, though, given that a user of our approach can easily filter out constraints that lead to false positives by selecting constraint components to avoid when inspecting the detection results. The results must also be seen in light of the fact that the underlying model collection varies in terms of quality (despite the automated quality checks we performed), abstraction-level, and modeling style. The insights from the individual logs suggest that our approach works particularly well for processes that are commonly executed in similar ways in organizations, which is exactly what we target by mining constraints from reference process models. Expectedly, for more specialized and niche processes the performance decreases.

Baseline comparison. Table 7.9 shows the results of the comparison against the baseline. The comparison results presented here focus on intra-object constraints, which the scope of the baseline is limited to. The baseline achieves an average precision of 0.80 and recall of 0.20. Our approach surpasses the baseline for all configurations in terms of recall: When choosing the configuration that yields the best trade-off between precision and recall ($k = 100, \tau = 0.5$), we achieve a slightly lower precision than the baseline (0.72), but a recall of 0.79, which is almost four times higher. Choosing a configuration with fewer constraints ($k = 10, \tau = 0.8$), we also surpass the baseline in terms of precision (0.86), while the recall we achieve with that configuration is still three times higher than the baseline's recall.

Our approach's improved performance, especially in terms of recall, can be attributed to its use of a broader range of constraints, resulting in an increase in true positives and a decrease in false negatives. In particular, the baseline focuses on detecting best-practice violations that depend on pairwise relations between actions that are applied to the same object, e.g. that *reject* and *accept* exclude each other.

The declarative constraint templates we use can also detect best-practice violations independent of such pairwise relations. For instance, we can detect that an order was never created independent of what happens to the order in the remainder of a trace, whereas the baseline can only detect this based on other actions applied to the order, e.g., if an *archive* action is applied to the order and it observed a relation *create* is followed by *archive*. Nevertheless, despite these improvements compared to the baseline, our approach detects a substantial number of false positives. Still, for the goal of detecting best-practice violations, our approach is generally better suited than the baseline. This is because we do not want users to miss potentially relevant best-practice violations; thus, it makes sense to sacrifice precision for recall. Only when we do this, we achieve a somewhat lower precision than the baseline, for the benefit of a much higher recall.

Table 7.9: Results of the baseline comparison only considering intra-object constraints (averaged across logs).

Approach	**TP**	**FP**	**FN**	**Precision**	**Recall**
Ours (best trade-off)	68.64	131.71	22.58	0.72	0.79
(best precision)	51.69	26.46	39.53	0.86	0.59
Baseline [5]	35.56	26.16	55.66	0.80	0.20

Run time. We ran our experiments single-threaded on a laptop with a 2 GHz Intel Core i5 processor and 16GB of memory. In this environment, the constraint mining stage, which has to be run only once for a model collection, took 112.5 minutes (99% of the time was used for constraint refinement). The average time to run the selection and checking stages for a log was 72.9 seconds, varying between 19.4 and 167.6 seconds depending on the complexity of the input event log, i.e., the number of activities and object types.

7.3.4 Application Cases

Finally, we applied our approach to real-world event data of a purchasing process and a sales process to highlight its usefulness in practical settings:

1. *Purchasing process*. We use the BPI19 event log [169], which captures event data on a purchasing process at a multinational company. We selected those traces from the original log that are supposed to adhere to a 3-ways-match procedure where the invoice is to be recorded after goods receipt. The resulting log contains 15,182 traces, 319,233 events, and 38 event classes.
2. *Sales process*. We use a proprietary event log of a sales process. The process instances were executed as part of a business-to-business software sales pro-

cess of a medium-sized European company. In this process, a lead (customer contact) is generated and eventually converted into an opportunity (lead with specific purchase scenario) that is then either closed or turned into a paying customer. This event log cannot be shared for privacy-related ethics and compliance reasons.

For the real-world application cases we employed a proprietary reference model collection of ca. 4,000 BPMN models that provide vendor-specific best practices.

Results. We discuss the results obtained for each process separately.

Purchasing process. Although there is no gold standard available that indicates best-practice violations in this process, our approach was able to detect a range of potentially undesired behaviors, as shown in Table 7.10.

ID	Constraint	Constraint explanation	#Traces
p1	RESPONSE(*create*,*confirm*)|*invoice*	After an *invoice* is *created*, it should be *confirmed*.	10,722
p2	SUCCESSION(*goods receipt*, *invoice*)	An *invoice* occurs if and only if it is followed by a *goods receipt*.	9,601
p3	ATLEASTONE(*create*)|*invoice*	An *invoice* must be *created*.	940
p4	RESPONSE(*purchase order item*, *goods receipt*)	A *goods receipt* should follow a *purchase order item*.	638

Table 7.10: Exemplary constraints that were selected by our approach and the number of traces that violated them in the purchasing event log.

The examples correspond to situations where an invoice is never created (p3), where purchase order items are never followed by goods receipt (p4), and invoices that are created but never confirmed (p1). All of these issues represent behavior that deviates from best practices extracted from a high-quality reference model collection that includes models for purchasing, invoicing, and payment processes. Most interestingly, in a considerable amount of traces a good receipt is not followed by an invoice (p2), which violates the underlying matching procedure, where invoices should be recorded after goods are received. These may include incomplete traces, yet, may also hint at undesired behavior, which is worth investigating further.

Sales process. For the sales process event log we also found some best-practice violations based on constraints recommended by our approach as shown in Table 7.4. In particular, there are traces where a lead is never created (s4) and assigned more than once or never (s1). Furthermore, there is a considerable amount of opportunities that are never closed (s2) and activities in the process that are recorded but not documented properly afterwards (s3). A discussion with a technical process operations specialist confirmed that all of these best-practice violations indicate

lacking process discipline in the affected traces, i.e., these do not adhere to established procedures, which may cause efficiency and quality issues in the process. The best-practice violations, therefore, provide valuable insights into process improvement opportunities for the organization.

ID	Constraint	Constraint explanation	#Traces
s1	EXACTLYONE(*assign*)| *lead*	A *lead* is *assigned* once.	220,057
s2	EXACTLYONE(*close*)|*opportunity*	An *opportunity* is *closed* once.	11,175
s3	ALTERNATERESPONSE(*activity logged task*, *enter notes*)	*Activity logged task* should be followed by *enter notes*.	9,228
s4	ATLEASTONE(*create*) | *lead*	For a *lead*, *create* should occur.	418

Table 7.11: Exemplary constraints that were selected by our approach and the number of traces that violated them in the sales process event log.

These insights further highlight the benefits of our approach to detect potentially undesired behavior without the need for dedicated process models to be available.

7.4 Limitations

Our evaluation results show that our approach accurately identifies best-practice violations in event logs and can thus provide valuable insights into data quality issues and potential conformance problems in the underlying process. Our approach and evaluation have to be considered in the light of certain limitations, though.

First, our approach does not cover constraints for all process perspectives, excluding constraints related to the time and data perspectives from consideration. For instance, it cannot cover that an order should be approved within a given amount of time or that a specific check should be performed for orders of a certain value. This is because such constraints are rarely found in reference process models, whereas they are also much harder to generalize than our activity-, object-, and role-related ones, because restrictions on time and monetary values can differ substantially across organizations and processes. Second, in order to provide relevant best-practice violations, the event log is assumed to be on the same granularity level as the components of constraints the approach mined. If the event log records activities on a more fine-granular level, it may, therefore, be necessary to first abstract the event log before applying our approach.

With respect to our evaluation, there are certain threats to validity. A threat to internal validity is that the process models used as a basis for our experiments, although being real-world models, stem from different sources and were established for different purposes, which means that they jointly do not represent a *reference*

model collection (due to the absence of publicly available reference collections). Despite automated quality checks, the contained models may thus vary in terms of quality, granularity, and modeling style. A threat to external validity is the (currently) limited application of our approach in practice. While we applied it on real-world event data, which yielded interesting best-practice violations that were verified with a specialist, it has not been field-tested in an organization so far.

7.5 Related Work

In this section, we discuss the primary research streams to which the work that is presented in this chapter relates: conformance checking, declarative process mining, process matching, and anomaly detection in process mining.

7.5.1 Conformance Checking

Conformance checking aims to detect deviations between true behavior recorded in event logs and desired behavior captured in a process model [41]. If the input model captures rules or regulations, conformance-checking techniques can thus be used for process compliance checking, where the goal is to detect violations of these [42]. Most research on conformance checking focuses on scenarios where an event log and a dedicated imperative process model, e.g., a BPMN diagram, are available. In this context, conformance is primarily checked based on so-called alignments [9, 17], where observed traces are compared with executions allowed by a predefined process model to find deviations.

In the context of declarative models, conformance checking involves checking for each trace in an event log, if it satisfies each constraint in a declarative process model [59]. To get a global view on conformance, the idea of aligning log traces to the closest *model trace* (considering all model constraints) can also be lifted to a declarative setting [52]. In our approach, we adopt the former approach, though, because we do not check conformance against a given normative (declarative) model. Instead, we mitigate the need for such a dedicated model and rather check against a collection of relevant, yet individual, best-practice constraints, essentially querying (cf. [143]) the event log for violating traces.

7.5.2 Declarative Process Mining

Declarative process mining is an active research field [59]. In this context, DECLARE [142] is among the most prominent formalisms. It relies on LTL, leveraging its reasoning mechanisms for, among others, model analysis and conformance checking, which we use in our approach. Research in this area is also dedicated to

identifying and resolving inconsistencies and redundancies in declarative process models [47, 48, 57], which we leverage for both refining our constraint collection in the *Constraint Mining* stage of the approach and for checking if the recommended constraints are actually consistent in the *Constraint Selection* stage.

The notion of *object-centric behavioral constraints* has recently been proposed. These constraints allow for a detailed specification of complex networks of objects and their relations that co-evolve in one or multiple processes [23]. While still in its infancy [59], this line of research is promising to be adopted for our approach. The level of detail this formalism can express goes beyond what is typically captured in BPMN diagrams, though. Therefore, we would first need to extend the scope of the input of our approach.

Research on declarative process mining further focuses on discovering declarative models from event logs [58, 121]. In this context, measuring the interestingness of traces of an event log given a declarative process model [45] and vice versa [44] is researched. This closely relates to the notion of relevance we employ in our approach, where we essentially want to assess how interesting a constraint is for a given log. Complementary to existing approaches, we consider semantic similarity as an indicator of interestingness.

7.5.3 Process Matching

Process matching techniques revolve around establishing links between process concepts found in various artifacts. The primary research focus lies in process model matching, where the goal is to find links between activities present in different process models. To achieve this, process model matchers leverage diverse process model features including model structure [61] and allowed behavior [105], but also natural-language-based features [75]. Beyond model-to-model matching, there is also work on matching (parts of) models to events recorded in event logs [2, 26]. The selection stage of our approach also creates links between constraint components and event log counterparts based on semantic similarity in order to select the most relevant constraints for the given log.

7.5.4 Anomaly Detection in Process Mining

Anomaly detection in process mining aims to identify anomalous process behavior in the traces of an event log. Unlike conformance-checking approaches, anomaly-detection approaches rely on the event data itself to detect anomalies [11].

Most approaches do this by identifying statistical outliers at the trace-level or the behavioral-relation-level. Basic approaches simply filter out a trace if its activity sequence is shared by less than a certain percentage of other traces in

the log [33]. Process-discovery approaches use such frequency-based detection strategies to filter out anomalies as well. For instance, the *Inductive Miner infrequent* [107] detects and eliminates infrequent directly-follows relations in order to preserve the most common process behavior. The resulting models of process discovery approaches can also be used for anomaly detection [11]. To this end, a model is discovered based on a subset of traces in a log, before conformance-checking between the full log and the discovered model is applied to identify anomalies. While most approaches focus on control-flow information, some of them also consider additional process perspectives. Böhmer and Rinderle-Ma [36] proposed an approach that considers multiple perspectives by calculating probabilities of event attributes to co-occur with certain activities. They further developed this approach and employ association rule mining to identify and explain multi-perspective anomalies [35]. Deep learning approaches use a neural network architecture to learn what constitutes proper process behavior across multiple perspectives and thus to learn to detect anomalies as well [134, 135, 136].

Recently, the notion of semantic anomaly detection has been proposed [5, 43], which aims to identify process behavior that does not make sense from a semantic perspective. We used a state-of-the-art approach [5] for this task as a baseline in our evaluation, showing that our approach outperforms it in detecting best-practice violations in both scope and accuracy.

7.6 Summary

In this chapter, we proposed an approach for mining declarative constraints from reference process models to detect best-practice violations in event logs. Our approach extracts and refines constraints based on imperative models, instantiates and selects relevant constraints given an event log, and checks whether the process executions recorded in the log violate the behavior captured by the constraints. In this manner, we mitigate the need for a process model that is specifically designed to capture the desired behavior of the process that is recorded in the given event log. The best-practice violations that our approach detects give insights into potential data quality issues and conformance problems. Regardless of their cause, these violations thus inform analysts about potentially problematic behavior contained in the log, which may distort process analysis results.

Our experiments show that with a small number of constrains, which our approach selects from a collection of tens of thousands of mined constraints, we correctly detect a substantial amount of best-practice violations. Furthermore, application scenarios based on real-world reference models and event logs demonstrate the approach's usefulness in practical settings.

Chapter 8

Conclusion

This chapter concludes this doctoral thesis. Section 8.1 provides a summary of the main results. Section 8.2 discusses the implications of its contributions for research and practice. Finally, Section 8.3 provides directions for future research starting from the work presented in this thesis.

8.1 Summary of the Results

In this thesis, we focused on the automated transformation of event data to close the gap between its characteristics and the data needs of particular analysis purposes. The main contributions of this thesis are provided by five approaches. We can summarize the results per contribution as follows:

- *Semantic annotation of event logs.* The application of certain process analysis techniques requires specific semantic information, such as actions, objects, and actors, to be associated with events of a given log. However, such information is often not readily available, which limits the process analysis options based on that log. In Chapter 3, we proposed an approach that addresses this, by identifying and categorizing semantic components in events and creating an annotated event log that makes them explicit. In this manner, it enables a broad range of semantics-aware process analysis options. We demonstrated our approach's efficacy through evaluation experiments using a wide range of real-world event logs. The results show that our approach accurately identifies the targeted semantic components from textual attributes, while our attribute classification techniques also yield good results when dealing with the information contained in non-textual attributes. In both cases, we showed that our approach outperforms state-of-the-art work and performs well in further categorizing the identified semantic components.

- *Constraint-driven abstraction of event logs.* Fine-granular events are prevalent in real-world organizational settings and lead to complex analysis results that do not provide useful insights into a process. To overcome this, event abstraction lifts fine-granular events to higher-level ones. However, existing approaches do not allow a user to specify what the abstracted event log should look like, which is crucial if they want to perform process analysis that has specific data needs. In Chapter 4, we proposed an event-log-abstraction approach that allows a user to impose requirements on the resulting log in terms of constraints. As such, the approach supports the specification of the properties that the resulting log should have, such that it is meaningful for a given analysis purpose. Our evaluation experiments using real-world event logs showed that our work considerably outperforms baseline techniques, whereas two application cases demonstrate its usefulness in practical settings.
- *Task-level-event recognition from user interaction data.* User interaction data provides detailed records about how a user performs their tasks in a process, even when this involves multiple applications. However, user interaction events cannot be used directly for process mining, because they neither indicate their relation to a process-level activity nor their relation to a specific process execution. In Chapter 5, we proposed an unsupervised approach for recognizing task-level events from user interaction data. It segments a stream of user interaction events to identify tasks, categorizes these according to their type, and relates tasks to each other via object instances it extracts from the user interaction events. In this manner, our approach creates events that meet the requirements of process mining settings. We demonstrated our approach's efficacy in an evaluation using real data and showed that it outperforms three streaming and two offline baselines.
- *Object-information extraction from event logs.* Object-centric event logs capture event data of processes that involve multiple concurrent object types, with potentially complex interrelations. Such logs allow process mining techniques to handle multi-object processes in an appropriate manner. However, event data is often not available in such a format. It is rather captured in case-centric event logs, which obscure the true relations between objects and events, causing data quality issues that lead to incorrect analysis results. In Chapter 6, we addressed this issue by proposing an approach that automatically transforms a case-centric event log into an object-centric one, which allows organizations to use their existing case-centric logs for object-centric process analysis. We demonstrated its efficacy in an evaluation showing that it accurately rediscovers object-centric from corresponding case-centric logs and that it can mitigate data quality issues in real-world logs.

- *Best-practice-violation detection in event logs.* Detecting undesired process behavior in event logs can reveal data-quality issues and conformance problems. It is primarily done based on dedicated process models that specify what is desired behavior. However, these are rarely available and their creation involves substantial effort. In Chapter 7, we proposed an approach that mitigates the need for dedicated models, by mining declarative constraints from reference process models to detect best-practice violations in event logs. Our approach extracts best-practice constraints from reference models, selects relevant constraints given an event log, and checks whether the process executions recorded in the log violate the behavior captured by the constraints. As such, the approach provides insights into potential data-quality and conformance problems. Our experiments showed that, with a small number of constrains that our approach selects from a collection of thousands of extracted constraints, it correctly detects a substantial amount of best-practice violations. An application on real-world reference models and event logs demonstrated the approach's usefulness in practical settings.

When developing and evaluating our approaches, we followed established practices from algorithm-engineering research (cf. Section 1.4). For each approach, we started from a real-world problem, derived one or more algorithmic tasks that jointly address this problem, designed algorithms that satisfy these tasks, and implemented them to evaluate the design. When evaluating the approaches we employed established evaluation metrics as well as real-world event data and accounted for relevant validity concerns. In this manner, we made substantial contributions to the body of knowledge of the process mining field.

8.2 Implications

This section discusses the implications of the work presented in this thesis, with Section 8.2.1 reflecting on implications for practice and Section 8.2.2 reflecting on implications for research.

8.2.1 Implications for Practice

The work presented in this thesis has several implications for organizations aiming to effectively analyze their processes based on available event data. Because our approaches close the gap between event data characteristics and the data needs of certain analysis purposes, we identify the following main implications for practice:

- *Enabling the application of existing process analysis techniques to previously unsuitable event data.* Three of our approaches transform event data

to make it usable with existing process analysis techniques, addressing the following use cases:
(1) Fine-granular event data poses a considerable challenge to organizations when analyzing their processes in a data-driven manner. Specifically, using such data for process mining directly yields complex results that do not provide useful insights into the process. Although event abstraction can alleviate this problem, existing approaches do not provide guarantees about the resulting data characteristics. We presented an approach that allows organizations to specify what characteristics an abstracted log should have (Chapter 4). In this manner, our approach enables purpose-driven process analyses on the basis of otherwise too fine-granular events.
(2) Some (parts of) processes are not supported by applications that have a specific relation to a process and, therefore, event data in the classical sense is not available for them. Consequently, such processes cannot be analyzed by process mining techniques. Although user interaction data has the potential to fill in the blanks in the process coverage of event data, it is unusable in process mining settings. We proposed an approach that transforms user interaction data into task-level events (Chapter 5). These fulfil the requirements of process mining settings and, thus, extend the coverage of event data that can be used for process analysis.
(3) Storing event data of a multi-object process in a case-centric log format causes data quality issues that, in turn, lead to an incomplete or incorrect picture of the true process. We proposed an approach that addresses this by transforming case-centric event logs of multi-object processes into object-centric logs (Chapter 6). The approach uncovers true relations in the process, while mitigating data quality issues that were introduced through a case-centric data representation. Thereby, it enables object-centric analysis on existing but previously unsuitable logs.

- *Enabling the application of novel analysis techniques that leverage semantic information.* Event data available to organizations frequently have characteristics that prevents them from analyzing their process from a semantic point of view, such as analyzing how different types of objects are handled in the process. The reason for this is that the required semantic information is not readily available in existing event logs. We address this by making semantic information about events explicit (Chapter 3), enabling a broad range of semantics-aware analysis options. Among others, the semantic components our approach identifies and categorizes enable organizations to analyze how particular objects are handled in their process and to analyze the automation degree of their process. Furthermore, it enables the semantics-aware abstrac-

tion of event logs (cf. Chapter 4), which groups together activities based on their meaning. In this manner, such abstraction provides a high-level view of processes based on "semantic stages" where, e.g., one stage may be concerned with processing information, whereas another is about making a decision in the process. Hence, our approach provides novel analysis options that were not yet available through existing process mining techniques.

- *Enabling the detection of problematic process behavior without the need for dedicated models.* Undesired process behavior in event logs poses a considerable threat to organizations. In particular, non-compliant process executions can substantially impact the efficiency of an organization and the quality of their process' output. Such behavior is commonly detected based on dedicated process models. However, these are often not available in organizations and are costly to establish from scratch. We present approach that detects best-practice violations in event logs based on constraints mined from reference process models (Chapter 7). Using our approach, organizations can detect undesired behavior without the need to establish dedicated models of their processes. Such violations may point to data quality issues and non-conforming process behavior. Therefore, their detection provides valuable information to organizations that aim to ensure efficient processes execution and high process output quality, with limited effort.

8.2.2 Implications for Research

Our contributions enable and inform several research directions in the process mining field. In particular, we identify implications of our results for research on *semantics-aware process mining*, *object-centric process mining*, *event abstraction*, and *stream-based process mining*:

- *Semantics-aware process mining.* Process mining has long focused on the analysis of the control-flow of processes on an abstract level, i.e., without considering the actual meaning of events. Our work on the semantic annotation of event logs (Chapter 3) and best-practice-violation detection (Chapter 7) shows that considering event meaning holds considerable potential. Our research thus serves as a starting point for the development of novel semantics-aware process mining techniques, such as conformance-checking techniques that can estimate the severity of conformance violations based on event semantics and discovery techniques that can infer unseen process behavior based on the meaning of observed activities.
- *Object-centric process mining.* Object-centric process mining has gained increasing attention in recent years and event log formats that support this

paradigm are (as of early 2024) being actively developed [30, 83, 85]. Our research on extracting object information from case-centric event logs (Chapter 6) has implications for research on object-centric process mining. This is especially the case for research on respective log formats. Among others, our work highlights the importance of a clear separation of object properties and event attributes as well as the need for representing object evolution and how individual events are involved in this evolution. Furthermore, our approach provides object-centric event logs that can serve as evaluation data for the research community and, therefore, contributes to the development of novel object-centric process mining techniques.

- *Event abstraction in process mining.* Abstracting fine-granular event data is an important task in process mining, among others, to alleviate the complexity of process analysis results. Existing research has, so far, neglected the fact that certain analysis purposes have specific data needs. We provide the research community with the first approach that abstracts event logs as much as possible, but guarantees characteristics of the resulting log that a user requires (Chapter 4). In this manner, our research provides a basis for the development of analysis techniques that build on data that provides such guarantees including, but not limited to, process discovery techniques.
- *Stream-based process mining.* Our research on recognizing task-level events from user interaction data (Chapter 5) transforms low-level events into higher-level ones based on an event stream. As such, our work demonstrates the potential of developing approaches that achieve similar goals based on other kinds of low-level data that is commonly available in a streaming setting, such as network traffic data [76] or sensor data [152].

8.3 Future Research

This thesis focused on the automated transformation of event data in order to provide organizations with a broad range of process analysis options based on the data available to them. There are several promising directions for future research that have not been addressed within this thesis, which open up additional possibilities for how organizations can use their event data to improve their processes. We see the greatest potential in this regard in extending semantics-aware process mining, further broadening the coverage of process mining, and providing organizations with explanations and recommendations regarding the characteristics of their event data based on our results.

- *Extending semantics-aware process mining.* The five approaches we developed all share that they consider the semantics of event data to transform

it. However, the work presented in this thesis is only the starting point of semantics-aware process mining. Taking the meaning of events into account can provide many more analysis opportunities in process mining. One such opportunity is the automated assessment of the severity of conformance violations. In particular, state-of-the-art conformance-checking techniques do not consider the severity of violations. For example, if an activity, e.g., *conduct fraud check* or *archive application*, is not executed, they cannot recognize that missing the fraud check is much more severe than not archiving an application. Semantics-aware conformance-checking may alleviate this shortcoming. A respective approach can then recognize the criticality of missing a fraud check compared to an archiving activity. To develop such an approach, the action categorization we propose in this thesis combined with recent advancements in NLP, such as large language models, and resources that capture common-sense knowledge about activities performed in organizations can serve as a starting point.

- *Further broadening process mining coverage.* Future research may continue to extend the parts of organizational processes that can be analyzed by process mining techniques. For instance, we have shown how recognizing task-level events based on user interaction data can fill-in blanks in the coverage of conventional event data used in process mining settings. Future research could focus on other types of low-level data, such as sensor data and network-traffic data. More importantly, future research should address the problem that, currently, the abstracted data still only covers a fragment of a process. In particular, the resulting task-level events only refer to the parts of the process supported by a user interface. More generally, the results of event-abstraction approaches are typically based on data from a single source and are detached from process-related data from other sources. Therefore, an important future research direction in this regard is the combination of event abstraction and data integration. The goal must be to support the automated integration of such fragmented event data and to jointly abstract them to a common granularity level that is meaningful for process analysis, and, thereby, provide the basis for a truly holistic view on processes.
- *From enabled analysis options to explanations and recommendations.* We have presented approaches that change event data characteristics to enable additional process analysis options. Future research can go beyond this and provide pointers to root causes of the misalignment between data characteristics and data needs. For instance, in the context of detecting best-practice violations, the cause of a frequently missing activity may be derived from the behavior of the process executions for which this occurs. In a next step,

approaches could even provide recommendations about how to address this misalignment. In particular, causes for data quality issues, missing data, and conformance violations could be identified and automated suggestions how to resolve these could be derived from them. The final step in this regard would be an action-oriented process mining setting [8], in which actions are derived and initiated automatically based on diagnostics obtained through analysis results. For instance, to address conformance issues (e.g., detected through best-practice violations), a notification is automatically sent to a manager based on identifying that employees repeatedly skip a mandatory check. To address data quality issues, changes to the recording procedure are recommended based on the results of applying event abstraction, or the recording is even automatically enhanced to get events at the right level of granularity in the first place. To implement this, traceability between the process, data recording, source systems, and event data is important. Therefore, also research on how to provide such traceability is needed.

Bibliography

[1] Han van der Aa, Claudio Di Ciccio, Henrik Leopold, and Hajo A Reijers. “Extracting declarative process models from natural language”. In: *International Conference on Advanced Information Systems Engineering*. Springer, 2019, pp. 365–382.

[2] Han van der Aa, Avigdor Gal, Henrik Leopold, Hajo A Reijers, Tomer Sagi, and Roee Shraga. “Instance-based process matching using event-log information”. In: *International Conference on Advanced Information Systems Engineering*. Springer, 2017, pp. 283–297.

[3] Han van der Aa, Henrik Leopold, and Hajo A Reijers. “Checking Process Compliance on the Basis of Uncertain Event-to-Activity Mappings”. In: *Intl. Conf. Advanced Inf. Syst. Engineering*. Springer, 2017, pp. 79–93.

[4] Han van der Aa, Henrik Leopold, and Hajo A Reijers. “Comparing textual descriptions to process models–The automatic detection of inconsistencies”. In: *Inf. Syst.* (2016).

[5] Han van der Aa, Adrian Rebmann, and Henrik Leopold. “Natural language-based detection of semantic execution anomalies in event logs”. In: *Information Systems* 102 (2021). Publisher: Elsevier.

[6] Wil M. P. van der Aalst. “Object-centric process mining: Dealing with divergence and convergence in event data”. In: *Software Engineering and Formal Methods*. Springer, 2019, pp. 3–25.

[7] Wil M. P. van der Aalst. *Process Mining - Data Science in Action, Second Edition*. Springer, 2016.

[8] Wil M. P. van der Aalst. “Process mining: a 360 degree overview”. In: *Process Mining Handbook*. Springer, 2022, pp. 3–34.

[9] Wil M. P. van der Aalst, Arya Adriansyah, and Boudewijn van Dongen. “Replaying history on process models for conformance checking and performance analysis”. In: *WIDM* 2.2 (2012). Publisher: John Wiley & Sons, Inc., pp. 182–192.

[10] Wil M. P. van der Aalst and Alessandro Berti. "Discovering object-centric Petri nets". In: *Fundamenta informaticae* 175.1-4 (2020), pp. 1–40.

[11] Wil M. P. van der Aalst and Ana Karla A de Medeiros. "Process mining and security: Detecting anomalous process executions and checking process conformance". In: *Electronic Notes in Theoretical Computer Science* 121 (2005). Publisher: Elsevier, pp. 3–21.

[12] Wil M. P. van der Aalst, Hajo A Reijers, and Minseok Song. "Discovering social networks from event logs". In: *Computer Supported Cooperative Work (CSCW)* 14.6 (2005). Publisher: Springer, pp. 549–593.

[13] Luka Abb and Jana-Rebecca Rehse. "A Reference Data Model for Process-Related User Interaction Logs". In: *Business Process Management*. Springer, 2022, pp. 57–74.

[14] Ziawasch Abedjan, Lukasz Golab, and Felix Naumann. "Profiling relational data: a survey". In: *The VLDB Journal* 24.4 (2015), pp. 557–581.

[15] Mehdi Acheli, Daniela Grigori, and Matthias Weidlich. "Discovering and Analyzing Contextual Behavioral Patterns from Event Logs". In: *IEEE Transactions on Knowledge and Data Engineering* Early access (2021).

[16] Mehdi Acheli, Daniela Grigori, and Matthias Weidlich. "Efficient Discovery of Compact Maximal Behavioral Patterns from Event Logs". In: 2019.

[17] Arya Adriansyah, Boudewijn van Dongen, and Wil M. P. van der Aalst. "Conformance Checking using Cost-Based Fitness Analysis". In: *EDOC*. IEEE, 2011, pp. 55–64.

[18] Simone Agostinelli, Andrea Marrella, Luka Abb, and Jana-Rebecca Rehse. "Mastering Robotic Process Automation with Process Mining". In: *Business Process Management*. Springer, 2022, pp. 47–53.

[19] Simone Agostinelli, Andrea Marrella, and Massimo Mecella. "Automated segmentation of user interface logs". In: *Robotic Process Automation*. De Gruyter Oldenbourg, 2021, pp. 201–222.

[20] Rakesh Agrawal and Ramakrishnan Srikant. "Mining sequential patterns". In: *Proceedings of the eleventh international conference on data engineering*. IEEE, 1995, pp. 3–14.

[21] Thomas Allweyer. *BPMN 2.0: introduction to the standard for business process modeling*. BoD–Books on Demand, 2010.

[22] John O. Aoga, Tias Guns, and Pierre Schaus. "Mining Time-Constrained Sequential Patterns with Constraint Programming". In: *Constraints* 22.4 (Oct. 2017). Publisher: Kluwer Academic Publishers, pp. 548–570.

[23] Alessandro Artale, Diego Calvanese, Marco Montali, and Wil MP van der Aalst. "Enriching Data Models with Behavioral Constraints." In: *Ontology Makes Sense* 316 (2019), pp. 257–277.

[24] Adriano Augusto, Raffaele Conforti, Marlon Dumas, Marcello La Rosa, Fabrizio Maria Maggi, Andrea Marrella, Massimo Mecella, and Allar Soo. "Automated discovery of process models from event logs: Review and benchmark". In: *IEEE Transactions on Knowledge and Data Engineering* 31.4 (2018), pp. 686–705.

[25] Ahmed Awad, Gero Decker, and Mathias Weske. "Efficient compliance checking using bpmn-q and temporal logic". In: *Intl. Conf. Business Process Management*. Springer, 2008, pp. 326–341.

[26] Thomas Baier, Jan Mendling, and Mathias Weske. "Bridging abstraction layers in process mining". In: *Inf. Syst.* 46 (2014). Publisher: Elsevier, pp. 123–139.

[27] Dorina Bano and Mathias Weske. "Discovering data models from event logs". In: *International Conference on Conceptual Modeling*. Springer, 2020, pp. 62–76.

[28] Marisol Barrientos, Karolin Winter, Juergen Mangler, and Stefanie Rinderle-Ma. "Verification of Quantitative Temporal Compliance Requirements in Process Descriptions Over Event Logs". In: *International Conference on Advanced Information Systems Engineering*. Springer, 2023, pp. 417–433.

[29] Iris Beerepoot, Daniël Barenholz, Stijn Beekhuis, Jens Gulden, Suhwan Lee, Xixi Lu, Sietse Overbeek, Inge van de Weerd, Jan Martijn van der Werf, and Hajo Reijers. "A Window of Opportunity: Active Window Tracking for Mining Work Practices". In: *ICPM*. IEEE, 2023.

[30] Alessandro Berti, István Koren, Jan Niklas Adams, Gyunam Park, Benedikt Knopp, Nina Graves, Majid Rafiei, Lukas Liß, Leah Tacke Genannt Unterberg, Yisong Zhang, et al. *OCEL (Object-Centric Event Log) 2.0 Specification*. 2023.

[31] Alessandro Berti, Gyunam Park, Majid Rafiei, and Wil M. P. van der Aalst. "A generic approach to extract object-centric event data from databases supporting SAP ERP". In: *Journal of Intelligent Information Systems* (2023). Publisher: Springer, pp. 1–23.

[32] Alessandro Berti, Sebastiaan J van Zelst, and Wil M. P. van der Aalst. "Process Mining for Python (PM4Py): Bridging the Gap Between Process-and Data Science". In: *ICPM Demo Track 2019*. 2019, pp. 13–16.

[33] Fábio Bezerra and Jacques Wainer. "Algorithms for anomaly detection of traces in logs of process aware information systems". In: *Information Systems* 38.1 (2013). Publisher: Elsevier, pp. 33–44.

[34] Albert Bifet, Ricard Gavalda, Geoffrey Holmes, and Bernhard Pfahringer. *Machine learning for data streams: with practical examples in MOA*. MIT press, 2018.

[35] Kristof Böhmer and Stefanie Rinderle-Ma. "Association rules for anomaly detection and root cause analysis in process executions". In: *Advanced Information Systems Engineering*. Springer, 2018, pp. 3–18.

[36] Kristof Böhmer and Stefanie Rinderle-Ma. "Multi-perspective anomaly detection in business process execution events". In: *On the Move to Meaningful Internet Systems*. Springer, 2016, pp. 80–98.

[37] Jan vom Brocke, Mieke Jans, Jan Mendling, and Hajo Reijers. "A five-level framework for research on process mining". In: *BISE* (2021).

[38] Joos Buijs. *Environmental permit application process ('WABO'), CoSeLoG project*. 2014. DOI: 10.4121/uuid:26aba40d-8b2d-435b-b5af-6d4bfbd7a270.

[39] Andrea Burattin. "Streaming process mining". In: *Process Mining Handbook, Springer* (2022). Publisher: Springer.

[40] Feng Cao, Martin Ester, Weining Qian, and Aoying Zhou. "Density-based clustering over an evolving data stream with noise". In: *International Conference on Data Mining*. SIAM, 2006, pp. 328–339.

[41] Josep Carmona, Boudewijn van Dongen, Andreas Solti, and Matthias Weidlich. *Conformance Checking*. Springer, 2018.

[42] Filip Caron, Jan Vanthienen, and Bart Baesens. "Comprehensive rule-based compliance checking and risk management with process mining". In: *Decision Support Systems* 54.3 (2013). Publisher: Elsevier, pp. 1357–1369.

[43] Julian Caspary, Adrian Rebmann, and Han van der Aa. "Does this make sense? Machine learning-based detection of semantic anomalies in business processes". In: *Business Process Management*. Springer, 2023, pp. 163–179.

[44] Alessio Cecconi, Giuseppe De Giacomo, Claudio Di Ciccio, Fabrizio Maria Maggi, and Jan Mendling. "Measuring the interestingness of temporal logic behavioral specifications in process mining". In: *Information Systems* 107 (2022). Publisher: Elsevier, p. 101920.

[45] Alessio Cecconi, Claudio Di Ciccio, Giuseppe De Giacomo, and Jan Mendling. "Interestingness of Traces in Declarative Process Mining: The Janus LTLp Approach". In: *BPM*. Springer, 2018, pp. 121–138.

[46] Shuo Chen, Josh L Moore, Douglas Turnbull, and Thorsten Joachims. "Playlist prediction via metric embedding". In: *SIGKDD*. ACM, 2012, pp. 714–722.

[47] Carl Corea, John Grant, and Matthias Thimm. "Measuring Inconsistency in Declarative Process Specifications". In: *Business Process Management*. Springer, 2022, pp. 289–306.

[48] Carl Corea, Sabine Nagel, Jan Mendling, and Patrick Delfmann. "Interactive and minimal repair of declarative process models". In: *BPM Forum*. Springer, 2021, pp. 3–19.

[49] Boris Cule, Len Feremans, and Bart Goethals. "Efficient discovery of sets of co-occurring items in event sequences". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 9851 LNAI (2016), pp. 361–377.

[50] Boris Cule, Bart Goethals, and Céline Robardet. "A new constraint for mining sets in sequences". In: *Proceedings of the 2009 SIAM international conference on data mining*. SIAM, 2009, pp. 317–328.

[51] Massimiliano De Leoni and Safa Dündar. "Event-log abstraction using batch session identification and clustering". In: *Proceedings of the ACM Symposium on Applied Computing* (2020), pp. 36–44.

[52] Massimiliano De Leoni, Fabrizio M. Maggi, and Wil M. P. van der Aalst. "An alignment-based framework to check the conformance of declarative process models and to preprocess event-log data". In: *Information Systems* 47 (2015), pp. 258–277.

[53] Massimiliano De Leoni and Felix Mannhardt. *Road Traffic Fine Management Process*. 2015. DOI: 10.4121/uuid:270fd440-1057-4fb9-89a9-b699b47990f5.

[54] Li Deng and Yang Liu. "A joint introduction to natural language processing and to deep learning". In: *Deep learning in natural language processing* (2018). Publisher: Springer, pp. 1–22.

[55] Amit V. Deokar and Jie Tao. "Semantics-based event log aggregation for process mining and analytics". In: *Information Systems Frontiers* 17.6 (2015), pp. 1209–1226.

[56] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *NAACL*. ACL, 2019, pp. 4171–4186.

[57] Claudio Di Ciccio, Fabrizio Maria Maggi, Marco Montali, and Jan Mendling. "Resolving inconsistencies and redundancies in declarative process models". In: *Information Systems* 64 (2017), pp. 425–446.

[58] Claudio Di Ciccio and Massimo Mecella. "On the discovery of declarative control flows for artful processes". In: *ACM Transactions on Management Information Systems (TMIS)* 5.4 (2015), pp. 1–37.

[59] Claudio Di Ciccio and Marco Montali. "Declarative Process Specifications: Reasoning, Discovery, Monitoring". In: *Process Mining Handbook*. Ed. by Wil M. P. van der Aalst and Josep Carmona. Vol. 448. Lecture Notes in Business Information Processing. Springer, 2022, pp. 108–152.

[60] Kiarash Diba, Kimon Batoulis, Matthias Weidlich, and Mathias Weske. "Extraction, correlation, and abstraction of event data for process mining". In: *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 10.3 (2020), pp. 1–31.

[61] Remco M Dijkman, Marlon Dumas, Boudewijn van Dongen, Reina Käärik, and Jan Mendling. "Similarity of business process models: Metrics and evaluation". In: *Inf. Syst.* 36.2 (2011). Publisher: Elsevier, pp. 498–516.

[62] Almir Djedović. *Credit Requirement Event Logs*. Sept. 2017. DOI: 10.4121/uuid:453e8ad1-4df0-4511-a916-93f46a37a1b5.

[63] Ivan Donadello, Francesco Riva, Fabrizio Maria Maggi, and Aladdin Shikhizada. "Declare4Py: A Python Library for Declarative Process Mining". In: *Proceedings of the Best Dissertation Award, Doctoral Consortium, and Demonstration & Resources Track at BPM* (2022), pp. 117–121.

[64] Boudewijn van Dongen. *BPI Challenge 2012*. 2012. DOI: 10.4121/uuid:3926db30-f712-4394-aebc-75976070e91f.

[65] Boudewijn van Dongen. *BPI Challenge 2014: Activity log for incidents*. 2014. DOI: 10.4121/uuid:86977bac-f874-49cf-8337-80f26bf5d2ef.

[66] Boudewijn van Dongen. *BPI Challenge 2015*. 2015. DOI: 10.4121/uuid:31a308ef-c844-48da-948c-305d167a0ec1.

[67] Boudewijn van Dongen. *BPI Challenge 2017*. 2017. DOI: 10.4121/uuid:5f3067df-f10b-45da-b98b-86ae4c7a310b.

[68] Boudewijn van Dongen. *BPI Challenge 2018*. 2018. DOI: 10.4121/uuid:3301445f-95e8-4ff0-98a4-901f1f204972.

[69] Boudewijn van Dongen. *BPI Challenge 2019*. 2019. DOI: 10.4121/uuid:d06aff4b-79f0-45e6-8ec8-e19730c248f1.

[70] Boudewijn van Dongen. *BPI Challenge 2020: Travel Permit Data*. 2020. DOI: 10.4121/uuid:ea03d361-a7cd-4f5e-83d8-5fbdf0362550.

[71] Nan Du, Kai Chen, Anjuli Kannan, Linh Tran, Yuhui Chen, and Izhak Shafran. "Extracting Symptoms and their Status from Clinical Conversations". In: *ACL*. 2019, pp. 915–925.

[72] Marlon Dumas, Marcello La Rosa, Jan Mendling, and Hajo A. Reijers. *Fundamentals of Business Process Management, Second Edition*. Springer, 2018.

[73] Maikel van Eck, Xixi Lu, Sander JJ Leemans, and Wil M. P. van der Aalst. "PM^2: a process mining project methodology". In: *International conference on advanced information systems engineering*. Springer, 2015, pp. 297–313.

[74] Maikel van Eck, Natalia Sidorova, and Wil van der Aalst. "Guided interaction exploration in artifact-centric process models". In: *Business Informatics*. IEEE, 2017, pp. 109–118.

[75] Marc Ehrig, Agnes Koschmider, and Andreas Oberweis. "Measuring similarity between semantic business process models". In: *Proc. fourth Asia-Pacific conference on Comceptual modelling-Volume 67*. 2007, pp. 71–80.

[76] Gal Engelberg, Moshe Hadad, and Pnina Soffer. "From Network Traffic Data to Business Activities: A Process Mining Driven Conceptualization". In: *BPMDS 2021*. Springer, 2021, pp. 3–18.

[77] Stefan Esser and Dirk Fahland. "Multi-dimensional event data in graph databases". In: *Journal on Data Semantics* 10.1 (2021), pp. 109–141.

[78] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. "A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise". In: *KDD*. AAAI Press, 1996, pp. 226–231.

[79] Bettina Fazzinga, Sergio Flesca, Filippo Furfaro, and Luigi Pontieri. "Process Mining meets argumentation: Explainable interpretations of low-level event logs via abstract argumentation". In: *Information Systems* 107 (2022). Publisher: Elsevier, p. 101987.

[80] Peter Fettke and Peter Loos. "Classification of reference models: a methodology and its application". In: *Information systems and e-business management* 1 (2003). Publisher: Springer, pp. 35–53.

[81] Francesco Folino, Massimo Guarascio, and Luigi Pontieri. "Mining multi-variant process models from low-level logs". In: *International Conference on Business Information Systems*. Springer, 2015, pp. 165–177.

[82] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. "Cypher: An Evolving Query Language for Property Graphs". In: *Proceedings of the 2018 International Conference on Management of Data*. SIGMOD '18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 1433–1445.

[83] Anahita Farhang Ghahfarokhi, Gyunam Park, Alessandro Berti, and Wil M. P. van der Aalst. "OCEL: A Standard for Object-Centric Event Logs". In: *New Trends in Database and Information Systems*. Cham: Springer International Publishing, 2021, pp. 169–175.

[84] Stijn Goedertier, Jan Vanthienen, and Filip Caron. "Declarative business process modelling: principles and modelling languages". In: *Enterprise Information Systems* 9.2 (2015). Publisher: Taylor & Francis, pp. 161–185.

[85] Alexandre Goossens, Johannes De Smedt, Jan Vanthienen, and Wil M. P. van der Aalst. "Enhancing data-awareness of object-centric event logs". In: *International Conference on Process Mining*. Springer, 2022, pp. 18–30.

[86] Florian Gottschalk, Wil M. P. van der Aalst, and Monique H Jansen-Vullers. "Mining reference process models and their configurations". In: *OTM Workshops*. Springer, 2008, pp. 263–272.

[87] Michael Grohs, Luka Abb, Nourhan Elsayed, and Jana-Rebecca Rehse. "Large Language Models can accomplish Business Process Management Tasks". In: *arXiv preprint arXiv:2307.09923* (2023).

[88] C. W. Günther and H. M. W. Verbeek. "XES standard definition". In: *IEEE Std* (2014).

[89] Christian W. Günther and Wil M. P. van der Aalst. "Fuzzy Mining – Adaptive Process Simplification Based on Multi-perspective Metrics". In: *Business Process Management*. Berlin, Heidelberg: Springer, 2007, pp. 328–343.

[90] Gurobi Optimization, LLC. *Gurobi Optimizer Reference Manual*. 2021. URL: `https://www.gurobi.com`.

[91] Moshe Hadad, Gal Engelberg, and Pnina Soffer. "From Network Traffic Data to a Business-Level Event Log". In: *International Conference on Business Process Modeling, Development and Support*. Springer, 2023, pp. 60–75.

[92] Cornelia Haisjackl, Irene Barba, Stefan Zugal, Pnina Soffer, Irit Hadar, Manfred Reichert, Jakob Pinggera, and Barbara Weber. "Understanding declare models: strategies, pitfalls, empirical results". In: *Software & Systems Modeling* 15.2 (2016). Publisher: Springer, pp. 325–352.

[93] Jiawei Han, Hong Cheng, Dong Xin, and Xifeng Yan. "Frequent pattern mining: current status and future directions". In: *Data mining and knowledge discovery* 15.1 (2007). Publisher: Springer, pp. 55–86.

[94] Marwan Hassani, Sergio Siccha, Florian Richter, and Thomas Seidl. "Efficient Process Discovery From Event Streams Using Sequential Pattern Mining". In: *SSCI*. IEEE, 2015, pp. 1366–1373.

[95] Luheng He, Kenton Lee, Mike Lewis, and Luke Zettlemoyer. "Deep semantic role labeling: What works and what's next". In: *ACL*. 2017, pp. 473–483.

[96] Arthur ter Hofstede, Wil M. P. van der Aalst, and Mathias Weske. "Business process management: A survey". In: *Business Process Management*. Springer, 2003, pp. 1–12.

[97] Matthew Honnibal and Ines Montani. "spacy 2: Natural language understanding with bloom embeddings, convolutional neural networks and incremental parsing". In: *To appear* 7.1 (2017).

[98] IBM. *Carbon Design System - Action Labels*. 2022. URL: `https://carbondesignsystem.com/`.

[99] R. P. Jagadeesh Chandra Bose and Wil M. P. van der Aalst. "Abstractions in Process Mining: A Taxonomy of Patterns". In: *Business Process Management*. Springer, 2009, pp. 159–175.

[100] Dan Jurafsky and James H Martin. *Speech & language processing*. Pearson Education India, 2000.

[101] Dan Jurafsky and James H Martin. *Speech & language processing*. 3rd ed. draft. 2023. URL: `https://web.stanford.edu/~jurafsky/slp3/`.

[102] Leonard Kaufman and Peter J Rousseeuw. *Finding groups in data: an introduction to cluster analysis*. Vol. 344. John Wiley & Sons, 2009.

[103] Paul R Kingsbury and Martha Palmer. "From TreeBank to PropBank." In: *LREC*. 2002, pp. 1989–1993.

[104] Finn Klessascheck, Tom Lichtenstein, Martin Meier, Simon Remy, Jan-Philipp Sachs, Luise Pufahl, Riccardo Miotto, Erwin P. Böttinger, and Mathias Weske. "Domain-Specific Event Abstraction". In: *International Conference on Business Information Systems*. 2021, pp. 117–126.

[105] Matthias Kunze, Matthias Weidlich, and Mathias Weske. "Behavioral similarity–a proper metric". In: *Business Process Management*. Springer, 2011, pp. 166–181.

[106] Sander J.J. Leemans, Kanika Goel, and Sebastiaan van Zelst. "Using Multi-Level Information in Hierarchical Process Mining: Balancing Behavioural Quality and Model Complexity". In: *2020 2nd International Conference on Process Mining (ICPM)*. 2020, pp. 137–144.

[107] Sander JJ Leemans, Dirk Fahland, and Wil M. P. van der Aalst. "Discovering block-structured process models from event logs containing infrequent behaviour". In: *Business Process Management Workshops*. Springer, 2014, pp. 66–78.

[108] Volodymyr Leno, Adriano Augusto, Marlon Dumas, Marcello La Rosa, Fabrizio Maria Maggi, and Artem Polyvyanyy. "Identifying candidate routines for robotic process automation from unsegmented UI logs". In: *2020 2nd International Conference on Process Mining (ICPM)*. IEEE, 2020, pp. 153–160.

[109] Volodymyr Leno, Artem Polyvyanyy, Marlon Dumas, Marcello La Rosa, and Fabrizio Maria Maggi. "Robotic process mining: vision and challenges". In: *Business & Information Systems Engineering* 63.3 (2021). Publisher: Springer, pp. 301–314.

[110] Henrik Leopold, Han van der Aa, Jelmer Offenberg, and Hajo A Reijers. "Using Hidden Markov Models for the accurate linguistic analysis of process model activity labels". In: *Information Systems* 83 (2019). Publisher: Elsevier, pp. 30–39.

[111] Henrik Leopold, Han van der Aa, and Hajo A Reijers. "Identifying candidate tasks for robotic process automation in textual process descriptions". In: *BPMDS*. Springer, 2018, pp. 67–81.

[112] Henrik Leopold, Jan Mendling, and Hajo A Reijers. "On the automatic labeling of process models". In: *Advanced Information Systems Engineering*. Springer, 2011, pp. 512–520.

[113] Henrik Leopold, Jan Mendling, Hajo A Reijers, and Marcello La Rosa. "Simplifying process model abstraction: Techniques for generating model names". In: *Information Systems* 39 (2014). Publisher: Elsevier, pp. 134–151.

[114] Henrik Leopold, Fabian Pittke, and Jan Mendling. "Automatic service derivation from business process model repositories via semantic technology". In: *Journal of Systems and Software* 108 (2015). Publisher: Elsevier, pp. 134–147.

[115] Beth Levin. *English verb classes and alternations: A preliminary investigation*. University of Chicago press, 1993.

[116] Guangming Li, Eduardo González López de Murillas, Renata Medeiros de Carvalho, and Wil M. P. van der Aalst. "Extracting Object-Centric Event Logs to Support Process Mining on Databases". In: *Advanced Information Systems Engineering*. Springer, 2018, pp. 182–199.

[117] Guangming Li, Renata Medeiros de Carvalho, and Wil M. P. van der Aalst. "Automatic discovery of object-centric behavioral constraint models". In: *Business Information Systems*. Springer, 2017, pp. 43–58.

[118] Christian Linn, Phileas Zimmermann, and Dirk Werth. "Desktop activity mining-a new level of detail in mining business processes". In: *Workshops der INFORMATIK 2018-Architekturen, Prozesse, Sicherheit und Nachhaltigkeit*. Köllen Druck+ Verlag GmbH, 2018.

[119] Xixi Lu, Dirk Fahland, Frank J.H.M. van den Biggelaar, and Wil M. P. van der Aalst. "Handling duplicated tasks in process discovery by refining event labels". In: *BPM*. 2016, pp. 329–347.

[120] Xixi Lu, Marijn Nagelkerke, Dennis Van De Wiel, and Dirk Fahland. "Discovering interacting artifacts from ERP systems". In: *IEEE Transactions on Services Computing* 8.6 (2015), pp. 861–873.

[121] Fabrizio M Maggi, RP Jagadeesh Chandra Bose, and Wil M. P. van der Aalst. "Efficient discovery of understandable declarative process models from event logs". In: *CAiSE 2012, Gdansk, Poland*. Springer, 2012, pp. 270–285.

[122] Thomas W Malone, Kevin Crowston, and George Arthur Herman. *Organizing business knowledge: The MIT process handbook*. MIT press, 2003.

[123] Felix Mannhardt. *Hospital Billing - Event Log*. 2017. DOI: 10.4121/uuid:76c46b83-c930-4798-a1c9-4be94dfeb741.

[124] Felix Mannhardt. *Sepsis Cases - Event Log*. 2016. DOI: 10.4121/uuid:915d2bfb-7e84-49ad-a286-dc35f063a460.

[125] Felix Mannhardt, Massimiliano de Leoni, Hajo A. Reijers, Wil M. P. van der Aalst, and Pieter J. Toussaint. "Guided Process Discovery – A pattern-based approach". In: *Information Systems* 76 (2018), pp. 1–18.

[126] Ronny Mans, Wil M. P. van der Aalst, Rob Vanwersch, and Arnold Moleman. "Process mining in healthcare: Data challenges when answering frequently posed questions". In: *Process Support and Knowledge Representation in Health Care*. Springer, 2013, pp. 140–153.

[127] Andrea Marrella and Tiziana Catarci. "Measuring the learnability of interactive systems using a Petri Net based approach". In: *Proceedings of the 2018 Designing Interactive Systems Conference*. 2018, pp. 1309–1319.

[128] Jan Mendling, Henrik Leopold, Henning Meyerhenke, and Benoît Depaire. *Methodology of Algorithm Engineering*. _eprint: 2310.18979. 2023.

[129] Jan Mendling, Hajo A Reijers, and Wil M. P. van der Aalst. "Seven process modeling guidelines (7PMG)". In: *Information and Software Technology* 52.2 (2010). Publisher: Elsevier, pp. 127–136.

[130] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. *Efficient Estimation of Word Representations in Vector Space*. _eprint: 1301.3781. 2013.

[131] George A Miller. "WordNet: a lexical database for English". In: *Communications of the ACM* 38.11 (1995). Publisher: ACM, pp. 39–41.

[132] Jorge Munoz-Gama, R. (Rene) de la Fuente, M. (Marcos) Sepúlveda, and R. (Ricardo) Fuentes. *Conformance Checking Challenge 2019 (CCC19)*. 2019. DOI: 10.4121/uuid:c923af09-ce93-44c3-ace0-c5508cf103ad.

[133] Sabine Nagel and Patrick Delfmann. "Investigating Inconsistency Understanding to Support Interactive Inconsistency Resolution in Declarative Process Models". In: *ECIS 2022, Timisoara, Romania*. 2022.

[134] Timo Nolle, Stefan Luettgen, Alexander Seeliger, and Max Mühlhäuser. "Analyzing business process anomalies using autoencoders". In: *Machine Learning* 107.11 (2018). Publisher: Springer, pp. 1875–1893.

[135] Timo Nolle, Stefan Luettgen, Alexander Seeliger, and Max Mühlhäuser. "Binet: Multi-perspective business process anomaly classification". In: *Information Systems* 103 (2022). Publisher: Elsevier, p. 101458.

[136] Timo Nolle, Alexander Seeliger, and Max Mühlhäuser. "Unsupervised anomaly detection in noisy business process event logs using denoising autoencoders". In: *International conference on discovery science*. Springer, 2016, pp. 442–456.

[137] Marco Pegoraro, Merih Seran Uysal, Tom-Hendrik Hülsmann, and Wil M. P. van der Aalst. "Resolving uncertain case identifiers in interaction logs: A user study". In: *arXiv preprint arXiv:2212.00009* (2022).

[138] Marco Pegoraro, Merih Seran Uysal, Tom-Hendrik Hülsmann, and Wil M. P. van der Aalst. "Uncertain case identifiers in process mining: A user study of the event-case correlation problem on click data". In: *International Conference on Business Process Modeling, Development and Support*. Springer, 2022, pp. 173–187.

[139] Jian Pei, Jiawei Han, Behzad Mortazavi-Asl, Jianyong Wang, Helen Pinto, Qiming Chen, Umeshwar Dayal, and Mei-Chun Hsu. "Mining sequential patterns by pattern-growth: The prefixspan approach". In: *IEEE Transactions on knowledge and data engineering* 16.11 (2004), pp. 1424–1440.

[140] Jian Pei, Jiawei Han, and Wei Wang. "Constraint-based sequential pattern mining: The pattern-growth methods". In: *Journal of Intelligent Information Systems* 28.2 (2007), pp. 133–160.

[141] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. "GloVe: Global Vectors for Word Representation". In: *EMNLP*. 2014, pp. 1532–1543.

[142] Maja Pesic, Helen Schonenberg, and Wil M. P. van der Aalst. "Declare: Full support for loosely-structured processes". In: *11th IEEE international enterprise distributed object computing conference (EDOC 2007)*. IEEE, 2007, pp. 287–287.

[143] Artem Polyvyanyy, Chun Ouyang, Alistair Barros, and Wil M. P. van der Aalst. "Process querying: Enabling business intelligence through query-based process analytics". In: *Decision Support Systems* 100 (2017), pp. 41–56.

[144] Viara Popova and Marlon Dumas. "Discovering unbounded synchronization conditions in artifact-centric process models". In: *Business Process Management*. Springer, 2013, pp. 28–40.

[145] Viara Popova, Dirk Fahland, and Marlon Dumas. "Artifact lifecycle discovery". In: *International Journal of Cooperative Information Systems* 24.01 (2015).

[146] Sameer Pradhan, Wayne Ward, Kadri Hacioglu, James H Martin, and Daniel Jurafsky. "Semantic role labeling using different syntactic views". In: *ACL*. 2005, pp. 581–588.

[147] Belen Ramos-Gutierrez, Angel Jesus Varela-Vaca, F. Javier Ortega, María Teresa Gomez-Lopez, and Moe Thandar Wynn. "A NLP-Oriented Methodology to Enhance Event Log Quality". In: *Enterprise, Business-Process and Information Systems Modeling*. Cham: Springer International Publishing, 2021, pp. 19–35.

[148] Adrian Rebmann and Han van der Aa. “Enabling semantics-aware process mining through the automatic annotation of event logs”. In: *Inf. Syst.* 110.102111 (2022).

[149] Adrian Rebmann and Han van der Aa. “Extracting semantic process information from the natural language in event logs”. In: *International Conference on Advanced Information Systems Engineering*. Springer, 2021, pp. 57–74.

[150] Adrian Rebmann and Han van der Aa. “Recognizing Task-level Events from User Interaction Data”. In: *Inf. Syst.* 102404 (2024).

[151] Adrian Rebmann and Han van der Aa. “Unsupervised Task Recognition from User Interaction Streams”. In: *International Conference on Advanced Information Systems Engineering*. Springer, 2023, pp. 141–157.

[152] Adrian Rebmann, Andreas Emrich, and Peter Fettke. “Enabling the discovery of manual processes using a multi-modal activity recognition approach”. In: *BPM Workshops*. Springer, 2019.

[153] Adrian Rebmann, Timotheus Kampik, Carl Corea, and Han Van der Aa. “Mining Constraints from Reference Process Models for Detecting Best-Practice Violations in Event Logs.” Paper under submission for Information Systems. 2023.

[154] Adrian Rebmann, Sönke Knoch, Andreas Emrich, Peter Fettke, and Peter Loos. “A multi-sensor approach for digital twins of manual assembly and commissioning”. In: *Procedia Manufacturing* 51 (2020). Publisher: Elsevier, pp. 549–556.

[155] Adrian Rebmann, Peter Pfeiffer, Peter Fettke, and Han van der Aa. “Multi-perspective Identification of Event Groups for Event Abstraction”. In: *ICPM Workshops*. Springer, 2022.

[156] Adrian Rebmann, Jana-Rebecca Rehse, and Han van der Aa. “Uncovering Object-centric Data in Classical Event Logs for the Automated Transformation from XES to OCEL”. In: *Business Process Management*. 2022.

[157] Adrian Rebmann, Matthias Weidlich, and Han van der Aa. “GECCO: Constraint-driven Abstraction of Low-level Event Logs”. In: *38th IEEE International Conference on Data Engineering*. IEEE, 2022.

[158] Jana-Rebecca Rehse and Peter Fettke. “Clustering business process activities for identifying reference model components”. In: *Business Process Management Workshops*. Springer, 2019, pp. 5–17.

[159] Hajo A Reijers and Jan Mendling. "A study into the factors that influence the understandability of business process models". In: *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans* 41.3 (2010). Publisher: IEEE, pp. 449–462.

[160] Hajo A Reijers, Tijs Slaats, and Christian Stahl. "Declarative modeling–an academic dream or the future for BPM?" In: *Business Process Management*. Springer, 2013, pp. 307–322.

[161] Nils Reimers and Iryna Gurevych. "Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks". In: *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Nov. 2019.

[162] Tony J. van Roy and Laurence A. Wolsey. "Solving Mixed Integer Programming Problems Using Automatic Reformulation". In: *Oper. Res.* 35.1 (Feb. 1987), pp. 45–57.

[163] Sareh Sadeghianasl, Arthur ter Hofstede, Suriadi Suriadi, and Selen Turkay. "Collaborative and Interactive Detection and Repair of Activity Labels in Process Event Logs". In: *ICPM*. 2020, pp. 41–48.

[164] Sareh Sadeghianasl, Arthur ter Hofstede, Moe Wynn, and Suriadi Suriadi. "A contextual approach to detecting synonymous and polluted activity labels in process event logs". In: *OTM*. Springer, 2019, pp. 76–94.

[165] Catherine Sai, Karolin Winter, Elsa Fernanda, and Stefanie Rinderle-Ma. "Detecting Deviations Between External and Internal Regulatory Requirements for Improved Process Compliance Assessment". In: *International Conference on Advanced Information Systems Engineering*. Springer, 2023, pp. 401–416.

[166] Erik F Sang and Fien De Meulder. "Introduction to the CoNLL-2003 shared task: Language-independent named entity recognition". In: *arXiv preprint cs/0306050* (2003).

[167] Diana Sola, Christian Warmuth, Bernhard Schäfer, Peyman Badakhshan, Jana-Rebecca Rehse, and Timotheus Kampik. "SAP Signavio Academic Models: A Large Process Model Dataset". In: *ICPM Workshops*. Springer, 2023, pp. 453–465.

[168] Robyn Speer, Joshua Chin, and Catherine Havasi. "Conceptnet 5.5: An open multilingual graph of general knowledge". In: *Thirty-first AAAI conference on artificial intelligence*. 2017.

[169] Ward Steeman. *BPI Challenge 2013, closed problems*. 2013. DOI: 10.4121/uuid:c2c3b154-ab26-4b31-a0e8-8f2350ddac11.

[170] Ava Swevels, Remco Dijkman, and Dirk Fahland. "Inferring missing entity identifiers from context using event knowledge graphs". In: *International Conference on Business Process Management*. Springer, 2023, pp. 180–197.

[171] Niek Tax, Benjamin Dalmas, Natalia Sidorova, Wil M. P. van der Aalst, and Sylvie Norre. "Interest-driven discovery of local process models". In: *Information Systems* 77 (2018). Publisher: Elsevier, pp. 105–117.

[172] Niek Tax, Natalia Sidorova, Reinder Haakma, and Wil M. P. van der Aalst. "Mining local process models". In: *Journal of Innovation in Digital Ecosystems* 3.2 (2016). Publisher: Elsevier, pp. 183–196.

[173] Niek Tax, Natalia Sidorova, Reinder Haakma, and Wil M. P. van der Aalst. "Mining Local Process Models with Constraints Efficiently: Applications to the Analysis of Smart Home Data". In: *Proceedings - 2018 International Conference on Intelligent Environments, IE 2018* (2018), pp. 56–63.

[174] Ghalia Tello, Gabriele Gianini, Rabeb Mizouni, and Ernesto Damiani. "Machine learning-based framework for log-lifting in business process mining applications". In: *Business Process Management*. Springer, 2019, pp. 232–249.

[175] Tin Truong-Chi and Philippe Fournier-Viger. "A survey of high utility sequential pattern mining". In: *High-Utility Pattern Mining*. Springer, 2019, pp. 97–129.

[176] Arava Tsoury, Pnina Soffer, and Iris Reinhartz-Berger. "A conceptual framework for supporting deep exploration of business process behavior". In: *ER*. Springer, 2018, pp. 58–71.

[177] Yuki Urabe, Sayaka Yagi, Kimio Tsuchikawa, and Haruo Oishi. "Task Clustering Method Using User Interaction Logs to Plan RPA Introduction". In: *BPM*. Springer, 2021, pp. 273–288.

[178] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. "Attention is all you need". In: *Advances in neural information processing systems* 30 (2017).

[179] Ulrike Von Luxburg. "A tutorial on spectral clustering". In: *Statistics and computing* 17.4 (2007). Publisher: Springer, pp. 395–416.

[180] Boris Wiegand, Dietrich Klakow, and Jilles Vreeken. "Mining Easily Understandable Models from Complex Event Logs". In: *Proceedings of the 2021 SIAM International Conference on Data Mining (SDM)*. 2021, pp. 244–252.

[181] Christopher Wilt, Jordan Thayer, and Wheeler Ruml. "A comparison of greedy search algorithms". In: *Proceedings of the 3rd Annual Symposium on Combinatorial Search, SoCS 2010* (2010), pp. 129–136.

[182] Karolin Winter, Han van der Aa, Stefanie Rinderle-Ma, and Matthias Weidlich. "Assessing the compliance of business process models with regulatory documents". In: *Conceptual Modeling*. Springer, 2020, pp. 189–203.

[183] Jing Xiong, Guohui Xiao, Tahir Emre Kalayci, Marco Montali, Zhenzhen Gu, and Diego Calvanese. "Extraction of object-centric event logs through virtual knowledge graphs". In: *35th International Workshop on Description Logics*. 2022.

[184] Junfu Yin, Zhigang Zheng, and Longbing Cao. "USpan: an efficient algorithm for mining high utility sequential patterns". In: *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*. 2012, pp. 660–668.

[185] Junfu Yin, Zhigang Zheng, Longbing Cao, Yin Song, and Wei Wei. "Efficiently mining top-k high utility sequential patterns". In: *2013 IEEE 13th international conference on data mining*. IEEE, 2013, pp. 1259–1264.

[186] Sebastiaan van Zelst, Mohammadreza Fani Sani, Alireza Ostovar, Raffaele Conforti, and Marcello La Rosa. "Filtering spurious events from event streams of business processes". In: *CAiSE*. Springer, 2018, pp. 35–52.

[187] Sebastiaan van Zelst, Felix Mannhardt, Massimiliano de Leoni, and Agnes Koschmider. "Event abstraction in process mining: literature review and taxonomy". In: *Granular Computing* 6.3 (2021). Publisher: Springer, pp. 719–736.

[188] Zhuosheng Zhang, Yuwei Wu, Hai Zhao, Zuchao Li, Shuailiang Zhang, Xi Zhou, and Xiang Zhou. "Semantics-aware BERT for language understanding". In: *AAAI*. Vol. 34, No. 05. 2020, pp. 9628–9635.

[189] Ziqi Zhang. "Effective and efficient semantic table interpretation using tableminer+". In: *Semantic Web* 8.6 (2017), pp. 921–957.

[190] Stefan Zugal, Jakob Pinggera, and Barbara Weber. "Toward enhanced lifecycle support for declarative processes". In: *Journal of Software: Evolution and Process* 24.3 (2012). Publisher: Wiley Online Library, pp. 285–302.